University of Alberta

Quantitative Software Engineering Methods for Quality Improvement

by

Snezana Djokic

©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment
of the requirements for the degree of Master of Science

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 2002

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-69699-5

Canada

University of Alberta

Library Release Form

Name of Author: Snezana Djokic

Title of Thesis: Quantitative Software Engineering Methods for Quality Improvement

Degree: Master of Science

Year this Degree Granted: 2002

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend them or sell such copies for private, scholarly or scientific purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Снежана Ђокић

Snezana Djokic
Apt# 3A, 9009-112th Street
Edmonton, AB
Canada T6G 2C5

FEBRUARY 27, 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis titled Quantitative Software Engineering Methods for Quality Improvement submitted by Snezana Djokic in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor, Dr. Witold Pedrycz

Committee Chair, Dr. Wilsun Xu

Committee Member, Dr. Petr Musilek

Committee Member, Dr. Eleni Stroulia

February 27, 2002

Abstract

Empirical studies in software engineering are performed in order to gain deeper understanding of software processes and draw conclusions on the basis of past experience.

This study investigates methods to model software data with the intention to further improve existing models. In particular, the focus of the research is on describing software defect proneness. Methods for generalizing regression models are proposed for the purpose of representing defect behavior with a single model across many software projects. In order to build correct models, meta-analytic study is used to investigate the quality of the available software metrics, evidencing that some of the metrics represent similar behavior of software. The issue of software defects is also addressed by combining empirical results from software inspection studies, demonstrating that some reading techniques for software inspection are more effective in defect detection than others. As an alternative approach to modeling software data, decision tree learning is applied to identify the most promising characteristics and extract rules for building prediction models. The results achieved show that such rules can be used to complement statistical methods in capturing defect behavior of software systems.

Acknowledgements

I would like to thank my supervisors, Dr. Witold Pedrycz and Dr. Giancarlo Succi for the guidance and invaluable advice throughout the research process. Thanks also go to Dr. Petr Musilek, Dr. Marek Reformat and Dr. James Miller for their precious help at different stages of my research.

The research contained within this thesis was supported by Research and Teaching Assistant position with the Electrical and Computer Engineering Department of the University of Alberta, Edmonton, Canada. This research and travel to conferences were also partly supported by the University of Alberta, the government of Alberta, the Natural Science Engineering Research Council of Canada, Alberta Software Engineering Research Consortium, and Nortel Networks.

I would also like to express my gratitude to Dr. Forrest Shull and the University of Maryland for providing the software inspection data for this research.

Special thanks go to Cheryl Card and Gautam Karnik for carefully reviewing the writing.

Finally, I would like to thank my family and friends for their constant and unconditional support of my work.

Table of Contents

1. Introduction - Software Engineering Discipline **5**

2. Software Metrics – Tools for Software Engineers **11**
- 2.1 Types of software metrics 15
  - 2.1.1 Structural metrics 17
  - 2.1.2 Object-Oriented metrics 20
    - 2.1.2.1 Object-oriented CK metrics suite 22
- 2.2 Software metrics scales 26
  - 2.2.1 Analyzing software measures assuming absolute values in narrow ranges 28

3. Data description **30**
- 3.1 Public domain data 30
- 3.2 Industrial data from telecommunication domain 31
- 3.3 NASA Ames data set 33
- 3.4 Simulated data sets 35
- 3.5 Software inspection data set 36
- 3.6 Data collection process 37

4 Modeling software data using software metrics **41**
- 4.1 Statistical models 41
  - 4.1.1 Presence of collinearity in multivariate regression models 47
- 4.2 Systems of regression equations 49
  - 4.2.1 Seemingly Unrelated Regression Equations 49
  - 4.2.2 Data-model method 52
- 4.3 Methods for Generalizing Results from Individual Studies 54
  - 4.3.1 Statistical meta-analysis 57
    - 4.3.1.1 Meta-analysis based on correlation coefficients 60
    - 4.3.1.1.1. Weighted Estimators of a Common Correlation 60
    - 4.3.1.2 Meta-analysis of effect sizes 63
    - 4.3.1.3 Controversy of meta-analytical approach 66
  - 4.3.2 Sign test 68
- 4.4 Decision trees 70
  - 4.4.1 Decision tree representation 73
  - 4.4.2 Designing the trees 76
  - 4.4.3 Optimal-sized tree 77
  - 4.4.4 Automatic generation of decision trees using software tools 78
- 4.5 Software Inspection 83
  - 4.5.1 Software inspection techniques 84
  - 4.5.2 Reading techniques for software inspection 87
  - 4.5.3 Experimental studies on software inspection 89
- 4.6 Validation of results 92

5. Empirical evaluation of the proposed quantitative software engineering methods **95**
- 5.1 Meta-analytic study of object-oriented systems 95
  - 5.1.1 Collinearity in software data 95
  - 5.1.2 Meta-analysis for identifying best predictor variables 98
- 5.2 Analyzing measures assuming low values on an absolute scale 100
- 5.3 Combining regression models across software projects 103
- 5.4 Application of decision trees to software data 109
- 5.5 Software inspection experiment 121

6. Conclusions and recommendations **124**

7. References **129**

List of Tables

Table 1: Overview of software characteristics as proposed by ISO/IEC 9126 7

Table 2: Roadmap of the undertaken empirical investigations 10

Table 3: Categorization of software metrics with examples 16

Table 4: Empirical studies validating CK metrics 24

Table 5: Software metrics scale types 28

Table 6: Example of collected measures on absolute scale 29

Table 7: Descriptive statistics for the number of classes 30

Table 8: Descriptive statistics for the number of classes and LOC across projects in two releases of industrial telecommunication data sets 32

Table 9: Description of NASA Ames avionics data set 34

Table 10: Description of available software inspection data 37

Table 11: Set of WebMetrics relations 39

Table 12: Summary of the metrics currently possible to extract using Web Metrics 40

Table 13: Generalizing results from individual studies using vote counting 56

Table 14: Exact values of the bias correction factor J(m) for effect sizes 64

Table 15: Possible variables in empirical studies on software inspection 89

Table 16: Summary of Weighted Correlations for public domain data sets 96

Table 17: Summary of Homogeneity Test for public domain data sets 96

Table 18: Summary of Weighted Correlations for industrial data sets 97

Table 19: Summary of Homogeneity Test for industrial data sets 98

Table 20: Summary of individual weighted correlations for each project 99

Table 21: Summary of correlations and homogeneity values 99

Table 22: Probabilities of obtaining values for DIT and NOC 102

Table 23: Results of the Sign Test for different values of the threshold 103

Table 24: Summary of correlations between errors for the first release of telecommunication data 104

Table 25: Summary of correlations between errors for the second release 104

Table 26: Regression models from the first release of telecommunication projects 105

Table 27: Regression models from the second release of telecommunication projects............................. 106

Table 28: Regression models based on training and testing approach 108

Table 29: Comparison of the two methods for combining regression models 108

Table 30: Validation of results across two releases of telecom projects.. 109

Table 31: Results of comparative experiment with different tree sizes 111

Table 32: Overview of tree sizes generated with S-PLUS for the telecom data set 112

Table 33: Success rates of extracted "best" rules from the telecom data set obtained using S-PLUS........ 114

Table 34: Summary of NASA Ames analysis as obtained with S-PLUS 117

Table 35: Comparison of performance of the best trees produced by CART... 118

Table 36: Comparison of two different-sized trees 119

Table 37: Comparison of the same-sized trees... 120

Table 38: Descriptive statistics for each application and each reading technique...................................... 122

Table 39: Summary of number of seeded defects for each project and application 122

Table 40: Study sizes and the corresponding bias correction factors 122

Table 41: Individual effect sizes and their weights 122

List of Figures

Figure 1: Levels of process maturity as proposed in Capability Maturity Model (adopted from Fenton and Pfleeger, 1997) 6

Figure 3: Costs to detect and repair errors in different stages of software development (adopted from Peters and Pedrycz, 2001) 11

Figure 4: GQM phases (taken from http://ivs.cs.uni-magdeburg.de/s/java/GQM) 13

Figure 5: The process of extracting and analyzing software metrics using WebMetrics 38

Figure 6: Unbiased estimator of effect size 64

Figure 7: Decision tree - example 73

Figure 8: Graphical representation of a decision tree 75

Figure 9: Example of automatically pruned tree 79

Figure 10: Optimized tree 79

Figure 11: Fagan inspection process 84

Figure 12: The Structured Walkthrough inspection process 85

Figure 13: Humphrey's inspection process 85

Figure 14: The inspection process as described in Gilb and Graham 86

Figure 15: Asynchronous inspection process 86

Figure 16: Active Design Review process 87

Figure 17: N-fold inspection process 87

Figure 18: Distributions of the maximum values of DIT and NOC 101

Figure 19: Illustrated regression models for the first release 105

Figure 20: Regression model obtained from the second release 106

Figure 21: Graphical representation of classification success 111

Figure 22: Full size tree generated with S-PLUS 112

Figure 23: Sample histogram for tree sizes 13 and 14 generated with S-PLUS 113

1. Introduction - Software Engineering Discipline

Software Engineering is a scientific discipline that embraces a whole myriad of methods and approaches with the same goal – to assist in the process of creating better software systems. Considering how software has become pervasive in all areas of everyday life, it is not difficult to understand the complex and increasingly important role of software engineering.

The term *engineering* in software projects refers to applying scientific principles, methods, models, standards and theories that make it possible to manage, plan, model, design, implement, measure, analyze, maintain, and evolve a software system. As an engineering discipline, software engineering recognizes existing organizational and financial constraints and looks for solutions within these constraints. It is not just concerned with the technical processes of software development but also with activities such as software project management and methods to support software production. Software engineering is concerned with all aspects of software production from early stages of system specification through to maintaining the system after its release.

Software engineering is a relatively young discipline, first proposed in 1968 (Sommerville, 2001). It emerged from early experience that an informal approach to software development was not enough for building complex software applications. This problem was referred to as a "software crisis" by Naur *et al.* (1976). Software projects were typically late, unreliable, difficult to maintain, and costing more than predicted. It was clear that new techniques and methods were needed to control the complexity of such complex systems

There are many books and papers devoted to the topic of software engineering. However, the field of software engineering is very wide, making it impossible to consider all aspects present in different software development processes. Hence, the literature dealing with this subject either attempts to describe all phases and aspects concisely (Sommerville, 2001; Pressman, 2001; Peters and Pedrycz, 2000; Pfleeger *et al.*, 2001), or focuses on some aspects and approaches in details (Firesmith, 1992; Bruegge and Dutoit, 2000).

With respect to the goals of software engineering, the quality achieved in software needs to be quantified. There are factors that need to be satisfied in order to state that the actual software product is 'good'.

One of the recognized criteria for measuring software quality is ISO 9126. The objective of this standard is to provide a framework for the evaluation of software quality. ISO 9126 does not provide requirements for software, but it defines a quality model that is applicable to every kind of software. It defines six product quality characteristics and provides a suggestion of quality sub-characteristics (Table 1).

Another widely accepted approach for assessing the maturity of a software development process is the *Capability Maturity Model* (CMM) proposed by the Software Engineering Institute (Software Engineering Institute, Carnegie Mellon University, 1995). It defines five levels of process maturity, ranging from ad hoc to repeatable, defined, managed and optimizing. The CMM model distinguishes one level from another in terms of key process activities going on at each level. This notion is illustrated in Figure 1.

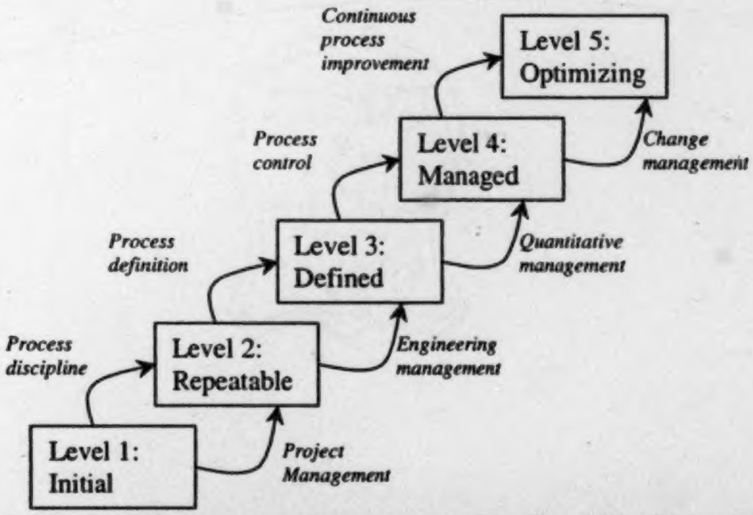
Continuous process improvement
Level 5: Optimizing
Process control
Level 4: Managed
Change management
Process definition
Level 3: Defined
Quantitative management
Process discipline
Level 2: Repeatable
Engineering management
Level 1: Initial
Project Management

Figure 1: Levels of process maturity as proposed in Capability Maturity Model (adopted from Fenton and Pfleeger, 1997)

Advancement in the ability to produce software has consequently led to higher complexity required to accommodate the increased requirements. New technologies and requirements in different application domains place new demands on software engineering.

Characteristics	Subcharacteristics	Definitions
	Suitability	Attributes of software that bear on the presence and appropriateness of a set of functions for specified tasks.
	Accurateness	Attributes of software that bear on the provision of right or agreed results or effects.
Functionality	Interoperability	Attributes of software that bear on its ability to interact with specified systems.
	Compliance	Attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions.
	Security	Attributes of software that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data.
	Maturity	Attributes of software that bear on the frequency of failure by faults in the software.
Reliability	Fault tolerance	Attributes of software that bear on its ability to maintain a specified level of performance in case of software faults or of infringement of its specified interface.
	Recoverability	Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it.
	Understandability	Attributes of software that bear on the users' effort for recognizing the logical concept and its applicability.
Usability	Learnability	Attributes of software that bear on the users' effort for learning its application.
	Operability	Attributes of software that bear on the users'effort for operation and operation control.
Efficiency	Time behavior	Attributes of software that bear on response and processing times and on throughput rates in performance of its function.
	Resource behavior	Attributes of software that bear on the amount of resource used and the duration of such use in performing its function.
	Analyzability	Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.
Maintainability	Changeability	Attributes of software that bear on the effort needed for modification, fault removal or for environmental change.
	Stability	Attributes of software that bear on the risk of unexpected effect of modifications.
	Testability	Attributes of software that bear on the effort needed for validating the modified software.
	Adaptability	Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered.
Portability	Installability	Attributes of software that bear on the effort needed to install the software in a specified environment.
	Conformance	Attributes of software that make the software adhere to standards or conventions relating to portability.
	Replaceability	Attributes of software that bear on opportunity and effort using it in the place of specified other software in the environment of that software.

Table 1: Overview of software characteristics as proposed by ISO/IEC 9126 (taken from http://www.iso.org/iso/pages)

Some of the software engineering techniques have become a part of software engineering and are now widely used. However, there are still problems in developing complex software that meets user expectations, is delivered on time and on budget. Many software projects still have problems and suggesting that software engineering is in a state of chronic affliction (Pressman, 2001). Consequently, there is a constant need for improvement.

The data describing software characteristics is typically hard to obtain and are of questionable quality therefore requiring careful assessment methods to model them. The objective of this research is to investigate methods that can provide legitimate models of software processes having in mind such characteristics of software data. This thesis deals with the following issues:

- Software defect behavior is examined from several aspects in this study. The study proposes methods for generalizing regression models describing defect behavior across projects in object-oriented systems. Seemingly Unrelated Regression Equations, the method applied in econometric studies to pool unique regression model out of many studies, shows potential to be applied to software projects in the similar fashion. In addition, the novel approach of optimizing regression across several models simultaneously on data and model level is applied in hope of providing a new dimension to this issue. The two methods are evaluated empirically on two consecutive releases of industrial data and their performances are compared.
- Characteristics of the object-oriented CK metrics (Chidamber and Kemerer, 1991) are examined, focusing on the presence of collinearity among them and the ranges of values they assume. To address the first issue, the meta-analytic technique of correlations between metrics is applied. The experiments are performed both on public domain and industrial data, confirming that in general these assumptions are correct. This means that in such metrics describe the same characteristics of the software, implying that not all of the extracted metrics are useful. Considering that the software metrics extraction process can be quite effort-consuming, this can lead to significant savings of time and resources. In addition, multivariate linear and log-linear prediction models produced using collinear variables can lead to ambiguous or incorrect results.

- Software defect behavior is also examined from the perspective of software inspection, a software engineering technique for early detection of software problems. A meta-analytic study is performed to compare and combine the effectiveness of different reading techniques for software inspection. The goal of this meta-analysis is to broaden the scope of empirical results beyond the scope of individual studies. For that purpose, the meta-analytic method of combining effect sizes is applied. The outcome is significant confirming higher effectiveness of one of the reading techniques in general.
- Decision tree learning is applied as an alternative approach of describing the characteristics of software systems. It is an exploratory method for describing the relationships among software attributes and its behavior. This research proposes devising such rules from decision trees since it presents a more flexible way to model software data. Since decision tree learning has been applied to software engineering studies to a very limited extent, this is a quite novel approach. Two different commercial data sets are used for the investigation of the applicability of decision trees for extracting such rules. The rules obtained from the analyses provide the information on the importance of predictors and offer prediction rules based on the past data. The results achieved in this research could be, when carefully assessed, used to form a knowledge base in both experimental and theoretical software engineering.

The thesis is organized as follows: Chapter 2 provides a systematic summary of the software metrics and associated statistical methods for measurement of different software attributes. All the data sets used in this research for empirical evaluation of quantitative software engineering methods are described in Chapter 3. A detailed overview of various data analysis techniques applied to software engineering data in this study is provided in Chapter 4. The potential of these models for quality improvement is empirically evaluated in Chapter 5, which outlines the results obtained from the applied methods and provides guidelines for interpreting results. Finally, Chapter 6 presents some conclusion reached and possible recommendations to improve related research.

Error! Reference source not found. shows the roadmap of the research carried out in this thesis where we visualize main links between the approaches (models), data sets, and results.

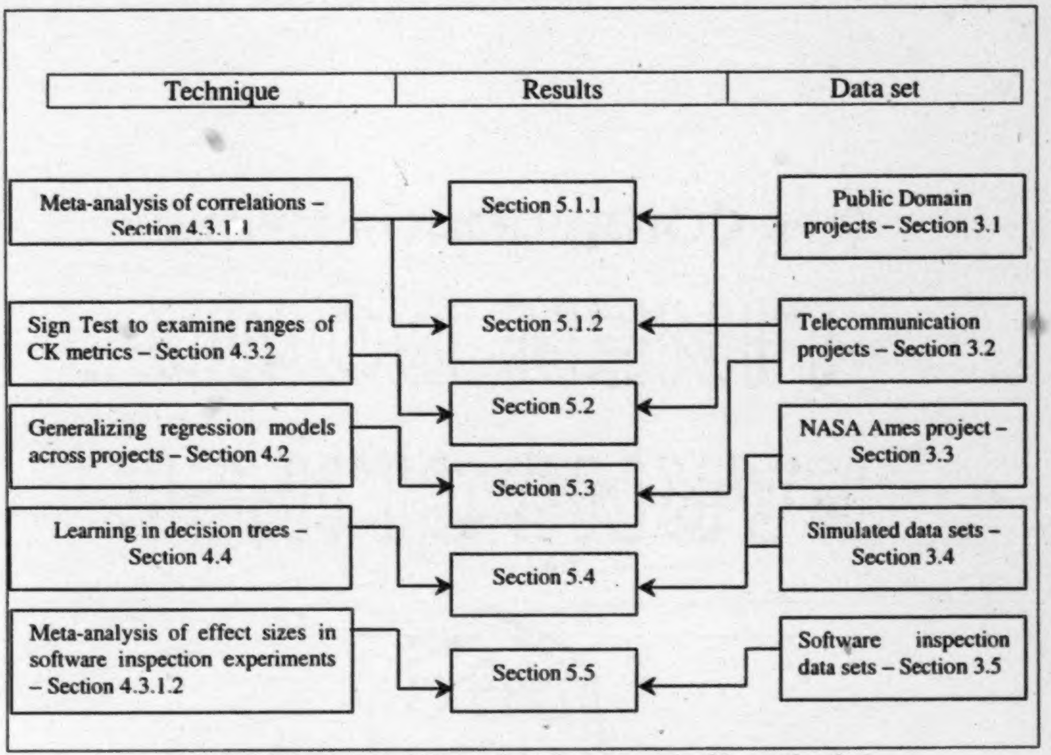
Technique
Results
Data set
Meta-analysis of correlations – Section 4.3.1.1
Sign Test to examine ranges of CK metrics – Section 4.3.2
Generalizing regression models across projects – Section 4.2
Learning in decision trees – Section 4.4
Meta-analysis of effect sizes in software inspection experiments – Section 4.3.1.2
Section 5.1.1
Section 5.1.2
Section 5.2
Section 5.3
Section 5.4
Section 5.5
Public Domain projects – Section 3.1
Telecommunication projects – Section 3.2
NASA Ames project – Section 3.3
Simulated data sets – Section 3.4
Software inspection data sets – Section 3.5

Table 2: Roadmap of the undertaken empirical investigations

2. Software Metrics – Tools for Software Engineers

As discussed in the previous section, the primary goal of software engineering research and practice is to improve quality of the developed software products. The purpose of software measurement is the extraction of quantifiable knowledge obtained from empirical studies. This quantification should enable a better control of the overall process of software development. Due to the experienced benefits of the software measurement approach, it currently presents one of the requirements for both higher levels of CMM and ISO 9126 standard.

Measurement evidences how the processes, products, resources, methods, and techniques of software development relate to one another. It can help answer questions about the effectiveness of techniques or tools, the productivity of development activities, the productivity of products and more.

A major goal of software measures is to capture the characteristics of the system under development as early as possible in the software development process (ideally in the analysis or design phase). The reason for this is that errors in predesign portion of a software engineering effort are less costly to correct than they are in later stages of a software project. Evidence of this can be seen in the distribution of the cost of repairing errors during each of the software process phases shown in Figure 2.

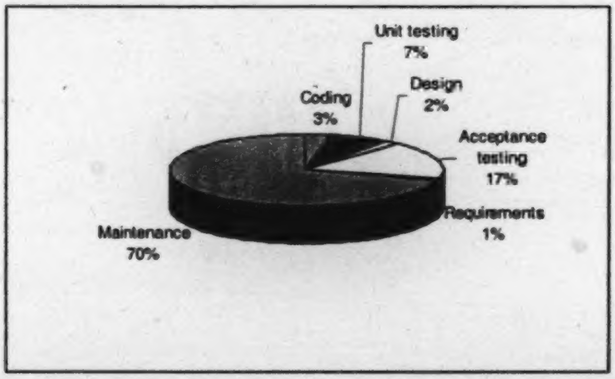

Figure 2: Costs to detect and repair errors in different stages of software development (adopted from Peters and Pedrycz, 2001)

The main focus of this study is to describe defect behavior of software systems. Several methods are applied to consider this issue. All the analyses are based on software metrics. Various software metrics are used depending on the availability and specific goals of each investigation. Some of them were extracted using the WebMetrics tool described later in this study, and some were readily available from the existing experiments.

Many metrics programs begin by measuring what is convenient or easy to measure, rather than by measuring what is needed. Such programs often fail because the resulting data are not useful to the developers and maintainers of the software. A measurement program can be more successful if it is designed with the goals of the project in mind.

Goal Question Metric (GQM) paradigm represents a mechanism for formalizing the characterization, planning, construction, analysis, and learning tasks in software engineering. It represents a systematic approach for setting project goals so that they are customized for a specific organization and defining them in an operational way.

Goal-oriented measurement points out that the existence of the explicitly stated goal is of the highest importance for improvement programs. Finally, the interpretation of the results gives the answer if the project goals were attained.

In other words, in order to improve the process, measurement goals need to be defined which will be, after applying GQM method, refined into questions and consecutively into metrics. The metrics will supply all the necessary information for answering those questions. The GQM method provides a measurement plan that deals with the particular set of problems and set of rules for obtained data interpretation. To clearly specify a set of operational goals, the measurement process is organized in a top-down order.

There are four basic steps in GQM approach:

1. Identification of goals
    a. define goals
    b. pose questions
    c. derive metrics
2. Plan measurement process
    a. identify hypotheses
    b. plan measurement
3. Perform measurement
    a. data collection

b. data validation

4. Analysis and interpretation

Figure 3 illustrates the sequence and relationships among different phases of GQM.

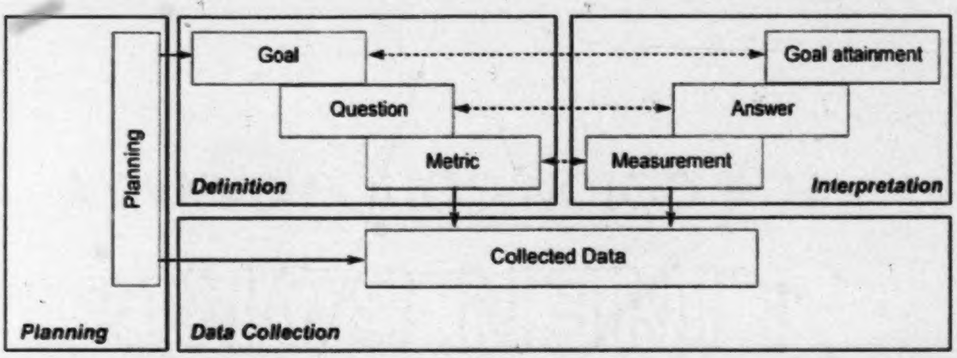

Figure 3: GQM phases (taken from http://ivs.cs.uni-magdeburg.de/s/java/GQM)

It is generally accepted that progressing from goal to question is the most difficult aspect of GQM. The method provides little guidance, relying instead on the judgment, experience and insight of those involved with measurement to identify useful questions.

There exists a multiplicity of questions that could be asked about virtually any goal.

First, the problem is choosing those questions likely to shed light, or to support achievement of that goal.

Second, the top-down nature of GQM can also lead to some difficulties. A goal may be so ambitious (for instance to eliminate all software defects prior to release), that it cannot readily be addressed by a manageable number of questions.

Another problem is ambiguous goals. Despite the template it is difficult to completely eradicate goals such as "improve software quality from the perspective of the development organization". In such circumstances it would be better to backtrack and refine the goal than to continue to try to identify measurement questions.

To alleviate problems in the process of defining these questions, different sets of guidelines exist for defining product-related and process-related questions in the GQM framework (Basili and Rombach, 1988). Product related questions are formulated for the purpose of defining the product attributes, such as cost, changes, or defects. They define a specific quality perspective of interest, such as reliability, for example. Process-related questions are formulated for the purpose of defining the specific quality perspective of the process quality, such as reduction of defects or cost effectiveness.

Given that metrics collection may be difficult, unrepeatable or expensive, it is a good idea to validate the proposed metric set, before proceeding to the next step of GQM. One effective way of accomplishing this is to build hypothetical profiles using the metrics that have been previously identified. The profiles can then be shown to an expert or target user, with a view to establishing whether they are able to meaningfully interpret the data without recourse to additional information.

In conclusion, a software metrics program can bring benefits to the software development process provided it is properly planned and implemented.

Many papers have focused on the topic of software measures. Some of them are fundamental, describing the theory of software measures and their validity (Morasca and Briand, 1997; Russell, 1992; Zuse and Bollmann-Sdorra, 1992; Baker *et al.*, 1990; Fenton, 1991). On the other hand, there are many experimental studies, attempting to apply these on real software systems and benefit from them (Bevan and Macloed, 1993; Bourque and Cot'e, 1990; Harrison, 1994; Kearney *et al.*, 1986; Khoshgoftaar and Allen, 1994; Miluk, 1993).

A number of different measures have been proposed and utilized in various projects, and a number of estimation techniques have been used. Still, the software remains somewhat elusive; excessive errors and wrong estimates are not uncommon, and a universally accepted set of measurement and estimation techniques has yet to be agreed upon. Therefore, a systematic approach towards the software measurement process needs to be taken.

The survey by Henderson-Sellers (1996) presents a large number of measures proposed by various authors, and introduces a multidimensional framework encompassing various categories of measures. A number of recommendations are given, although a general agreement has yet to be reached, as the evidence available is still, for the most part, inconclusive.

The study by Wang (2001) presents a software engineering measurement system (SEMS) for object-oriented systems. The proposed system currently consists of 304 software measures, which have been proposed by different studies, and aims at organizing existing and introducing new object-oriented software measures. The authors propose creating a system of software measures that would contain both *meta* and derived measures. By meta measures, the author means basic measures, from which all the other measures should be derived. The architecture of object-oriented systems can be divided into three levels namely *system*, *class* and *method*. Since the commonly used measures are focused on the lower level, this study proposes that measures should capture phenomena of interest at all the mentioned levels.

The importance of creating a systematic way of collecting software metrics is also discussed in (Kitchenham *et al.*, 2001). Organizing and maintaining software measures collected from various experiments are an important issue for two reasons. First, it would allow applying meta-analysis methods to assess whether phenomena of interest apply generally or are context dependent. Second, the proper data collection would enable creating data sets large enough to allow investigation of the many factors that can affect software project performance.

The following sections introduce different types of software metrics, and the issue of software metrics scales.

2.1 Types of software metrics

Measurement is performed on software development process, various software products, and resources. Therefore, different metric types are suited to each.

A *process* represents a collection of software-related activities. A *product* is any artifact that results form a process activity. *Resources* are entities required by a process activity.

Within each class of entity, there are internal and external attributes. Internal attributes of a product, process, or resource are those that can be measured purely in terms of the product, process, or resource itself, separate from its behavior. External attributes of product, process, or resource are those that can be measured only with respect to how the product, process, or resource relates to its environment.

An example of a product attribute is design. Some examples of internal aspects of design are size, coupling, complexity, and cohesiveness. External aspects are quality, complexity for use, and extendibility.

Examples of internal attributes of the testing process are time, effort, and number of discovered problems or defects, while external attributes are cost and cost-effectiveness.

Internal attributes of human resources in software development are effort and utilization, and external aspects are productivity, experience, and satisfaction.

Product metrics are concerned with characteristics of the software itself. There are two classes of these metrics: dynamic and static.

Dynamic metrics are collected by measurements made of a program in execution, while static metrics are collected by measurements made of the system representations such as design, code, or documentation.

These different types of metrics are related to different quality attributes. Dynamic metrics help to assess the efficiency whereas static metrics help to assess the complexity.

In addition to the objective metrics, subjective metrics are also required in some cases, especially for aspects such as experience of personnel, type of application, understandability etc.

Some examples of different types of software measures are shown in Table 3.

Criterion	Types	Example
Measured independently of behavior towards environment	Internal	Size, coupling, complexity
	External	Cost, correctness, cost-effectiveness
Whether it is related to the program's execution	Static	Execution time
	Dynamic	Cyclomatic complexity
Whether a measure is repeatable in exactly the same way	Objective	Number of discovered defects
	Subjective	Understandability, type of application

Table 3: Categorization of software metrics with examples

With a variety of the metrics proposed in software engineering, there is a clear need for validation whether a specific measure captures the attributes it claims to describe (Weyuker, 1988). Although it is essential to validate the characteristics of a measure, it is also important to determine whether the measure is a part of a valid prediction system, that is, to demonstrate that the measure is useful for estimation and prediction of some dependant variable in the software development process (Fenton and Pfleeger, 1997). This means that a measure must be viewed in the context in which it will be used; a measure may be valid for some uses but not for others. However, a measure can serve only one of these purposes, and it should not be rejected as invalid just because it is not a predictor.

It is said that a measure is valid in the narrow sense (internally valid) if it is useful for assessment purposes. If the measure is valid in the narrow sense, it is also called internally or semantically valid. If a measure is also a component of a prediction system, it is said to be valid in the wide sense (Fenton and Pfleeger, 1997).

The best way to validate a software measure is to use it on multiple data sets and assess its usefulness for description of measured attributes or as a component of a prediction system. In this study, the CK metrics suite is used to capture defect behavior of the classes in the system, and some other measures are used to model phenomena related to software problem reports.

Another aspect from which software measures should be regarded is whether and to what extent they suit a particular methodology of software development. The following section outlines two major categories of measures with respect to this issue.

2.1.1 Structural metrics

In late 70's and early 80's, structured methods introduced a more structured approach to software design. Since there was a clear need for formal design and change control, structured methods represent sets of notations and guidelines for software design. Examples include Structured Design (Constantine and Yourdon, 1979), Jackson System Development (Jackson, 1983), and various approaches to object-oriented design (Booch, 1994; Rumbaugh *et al.*, 1991; Jacobson, 1992).

There are many structural metrics proposed in literature and practice. One of the most obvious and easiest to understand internal software product attributes is the size of the software system. The size is a static attribute that can be measured without having to execute the system. However, even for this relatively simple attribute, there are multiple ways to measure it regarding different perspectives of interest.

There are three major products of the software development process whose size can be measured: specification, design, and code. The commonly used measure of source code program length is the number of Lines Of Code (LOC). Many different schemes have been proposed for counting LOC. A simple and widely accepted way of counting LOC based on number of semicolons is used in this study. An alternative measure of source code length is *Thousands of Lines of Code* or KLOC. In case of larger programs this measure can be used in just the same manner as LOC. All operations valid for LOC are also applicable to KLOC.

An alternative measure of size is the amount of functionality in the system. For measuring the amount of functionality in a system, function points are suggested by Albrecht and Gaffney (1986). Function points are intended to measure the amount of functionality in a system as described by a specification. In that way, they can be computed without forcing the specification to conform to the prescripts of a particular specification model or technique.

Function points include external inputs, external outputs, user inquires, external files and internal files. Each of them is then assigned a subjective complexity or a weight in the overall complexity. Summing the products of each function points and its weighting factor results in an *unadjusted function point count* (UFC). Finally, the adjusted function point count (FP) is obtained by multiplying the UFC with the *technical complexity factor* (TCF) as explained in (Fenton and Pfleeger, 1997).

Although related to size, complexity represents another interesting product attribute that can be interpreted in different ways. For example, Fenton and Pfleeger (1997) refer to the computational complexity, algorithmic complexity, structural complexity, and cognitive complexity. Computational complexity reflects the complexity of the underlying problem. Algorithmic complexity measures the complexity of the algorithm used to solve the problem. Structural complexity is used to quantify the structure of the software that implements the algorithm. Cognitive complexity measures the effort required to understand the software implementation.

Different ways of measuring complexity have been proposed. One of the simple yet useful techniques has been proposed by McCabe (1976). He proposes measuring software complexity by the cyclomatic number of the program's flow graph. For a program with the flow graph *F*, the cyclomatic number *C* measures the number of linearly independent paths through *F* and can be calculated as:

$$C(F) = a - n + 2$$

where *F* has *a* arcs and *n* nodes. Although the cyclomatic number cannot be used as a general complexity measure (Fenton and Pfleeger, 1997), sometimes it is a useful indicator of the maintenance effort.

On the other hand, Halstead (1982) attempts to capture attributes of a program that parallels physical and psychological measurements in other disciplines. He begins by defining a program *P* as a collection of tokens, classified as either operators or operands.

The length of *P* is defined as $N = No_1 + No_2$, while the *vocabulary* of *P* is $o = o_1 + o_2$. The *volume* of a program, described as the number of mental comparisons needed to write a program of length *N*, is

$$V = N\log_2 o$$

where o_1 is the number of unique operators, o_2 is the number of unique operands, No_1 is the number of unique operators and No_2 number of unique operands.

The volume can be used as a good unit measure, since it is not related to the type of implementation. Therefore, other implementations of the same algorithm can be compared.

In that sense, the ***program level*** of a program *P* of volume *V* is defined as the ratio

$$L = V^*/V$$

where V^* is the potential volume – the volume of the minimal size implementation of *P*. In general, $V<V^*$; therefore L is below 1.

The inverse of program level is the *difficulty*:

$$D = 1/L$$

For low-level languages, the resulting volume is high, thus leading to an increased difficulty of implementation. This means that V^* is quite difficult to estimate. Therefore, an alternative estimate for the program level is proposed as

$$\hat{L} = \frac{1}{D} = \frac{2}{o_1}\frac{o_2}{No_2}$$

The rationale behind this formula is that if the ratio of the numbers of operators increases, it is natural that the level of difficulty increases, as well. Also, if some operands are used repetitively, that contributes to the increase of the difficulty level.

Likewise, the estimated program length is defined as

$$\hat{N} = o_1 \log_2 o_1 + o_2 \log_2 o_2$$

Consequently, the effort required to generate *P* is then given by

$$\mathrm{E} = \frac{\mathrm{V}}{\hat{\mathrm{L}}} = \frac{\mathrm{o}_1 \mathrm{N} \mathrm{o}_2 \mathrm{N} \log_2 \mathrm{o}}{2\mathrm{o}_2}$$

Halstead's measurement system has been criticized in many studies; however, because of its simplicity it can present a rough estimate on certain occasions. The complexity metrics proposed by Halstead's Software Science show a sound background with useful insights into the nature of programming activities. They are focused on the code, and can be used to predict some software characteristics such as cost, development time, and reliability.

2.1.2 Object-Oriented metrics

In early 90's a new development methodology has been introduced – object-oriented design. One of the main reasons for moving to the object-oriented paradigm for developing complex applications is that it allows designers to model the real world more intuitively (Riel, 1996).

In the definition of the object-oriented systems by Yand and Weber (1990), the world is composed of substantial individuals that possess a finite set of properties. Collectively, an individual and its properties constitute an object. A class is a set of objects with common properties. Attributes such as coupling, cohesion, inheritance, and object complexity are defined for the classes of an object-oriented system.

Object-oriented methods include an inheritance model of the system, models of the static and dynamic relationships between objects and model of objects interactions at run time.

The object-oriented life-cycle model describes software development in terms of the identification of objects used to construct objects and object networks. The object-oriented model prescribes software development in terms of a synergy between abstraction, modularity, encapsulation, hierarchy, typing, concurrency, and persistence.

The increased presence and success of the object-oriented methods caused a need for appropriate object-oriented measures, since the existing structured metrics were not completely transferable to this new software methodology. Many object-oriented software metrics have been proposed and validated both theoretically and with respect to their applicability in practice.

Chidamber and Kemerer (1991) proposed a set of six metrics for object-oriented design. Since this metrics suite is extensively used in this research, they will be described in details in Section 2.1.2.1.

Lorenz and Kidd (1994) propose a number of possible object-oriented software measures, grouped in two broad categories of project and design measures. They suggest that managers and designers should set thresholds for the measures, and adjust them from project to project. Although the recommendations are not always clear and the study is more oriented towards Smalltalk, it still has some practical value.

Fioravanti and Nesi (to appear) use a wide range of object-oriented metrics to assess projects from a managerial perspective. Some 53 metrics are extracted using the proposed proprietary tool (TAC++) and different implemented views supported by the tool are suggested to assess them.

Mišić and Tešić (1997) use object-oriented metrics to estimate effort and complexity in software systems. The choice of metrics used is mainly focused on the measures available at the end of the design phase, as they provide the early input for effort and quality estimation. Metrics are grouped into two categories: class model measures and source code measures. In the first group are: Number of classes, Number of classes without superclass, Number of root classes, Number of methods, Number of methods for base classes. In the second category the following metrics are chosen: LOC including comments, Number of non-comment lines, Number of root classes, Number of base classes, Total number of functions etc.

Briand *et al.* (2000b) investigate a total of 28 coupling measures, 10 cohesion measures, 11 inheritance measures and a small selection of size measures. The goal of the study is to build prediction systems that would describe fault-proneness of object-oriented software systems. Different statistical models with selected metrics are built and evaluated based on these measures to achieve this goal.

El Emam *et al.* (2001) consider a total of 24 object-oriented metrics. The study selects a subset of these measures based on the statistical significance of the parameters in the logistic models. Also, attention is paid to the collinearity between the metrics, and only the orthogonal ones are selected. Finally, six object-oriented metrics are chosen for building the prediction models; two ancestor-based coupling measures (OCAEC and OCAIC), two descendent-based coupling metrics (OCMEC and OCMIC), and two metrics from the CK suite (DIT and NOC).

2.1.2.1 Object-oriented CK metrics suite

A relatively simple and well-understood set of CK metrics is used extensively in this study. When available, this suite of six metrics shows a good potential as a complete measurement framework in an object-oriented environment.

This research uses this set of metrics together with the Lines of Codes measure, for these measures were available for extraction using WebMetrics tool described in more details later in the study. Object-oriented metrics from this suite are explained in details here.

Depth of Inheritance Tree (DIT) for a class corresponds to the maximum length from the root of the inheritance hierarchy to the node of the observed class. It tries to capture properties originating from the inheritance relation. In the case of multiple inheritance, Depth of Inheritance Tree is the maximum length from the node to the root.

Another metric related to inheritance is the number of children (NOC), representing the number of immediate descendants of the class in the inheritance tree.

Coupling Between Objects (CBO) is defined as the number of other classes to which a class is coupled through method invocation or use of instance variables. CBO relates to the notion that an object is coupled to another object if one of them acts on the other. One of the good practices in software engineering, modularity, is achieved when coupling between classes is minimized.

Response For a Class (RFC) is the cardinality of the set of all internal methods and external methods directly invoked by them.

Weighted Methods per Class (WMC) measure is a sum of the complexities of the methods. The higher the value of WMC, the more complex the class is. This metric focuses on methods in a class and not on the interaction between classes. Usually, the number of methods (NOM) is used as a simplified version of the more general weighted methods count (WMC). The number of internal methods is extracted instead of forming a weighted sum of methods based on complexity.

Lack of Cohesion in methods (LCOM) concentrates on cohesion between the methods used by a class. It is defined as the number of pairs of non-cohesive methods minus the count of cohesive method pairs, based on common instance variables used by the methods in a class. The LCOM measure reveals the disparate nature of methods in the class. If the value of this measure is high, it implies that the class should be split into two or more subclasses.

Object-oriented software measures from the CK suite have been validated empirically in many studies. Table 4 outlines some of the studies with their conclusions on CK metrics, and their detailed descriptions are provided later in the text.

Li and Henry (1993) examine the usefulness of five out of the six CK metrics in predicting effort in software maintenance. The study does not include CBO in the investigation, because it is not related to inheritance. The empirical validation is conducted on two commercial systems using multivariate linear regression. The initial multivariate model contains ten predictors, including five CK metrics, two size measures, Number of Methods, Data Abstraction Coupling and Message-passing Coupling. Two size measures are later discarded because of the confirmed collinearity with other measures, therefore the final model includes eight regressors. The results show that these measures can be employed to predict maintenance effort in object-oriented systems, measuring number of lines changed in each class.

Basili *et al.* (1996) investigate if CK measures could provide relevant insight into fault-proneness of classes. The study is based on the data collected in a university setting from eight medium-sized information management systems developed with the same requirements. Univariate regression models are used with each CK metric. The results indicate that in general, DIT, RFC, NOC and CBO are very significant, WMC is somewhat significant, and LCOM was shown to be insignificant in all cases.

Study	Conclusions
Li and Henry, 1993	Study conducted on two commercial systems shows that five out of six CK metrics are (without CBO) useful for predicting effort in software maintenance.
Basili *et al.*, 1996	Eight medium size projects in university setting show that DIT, RFC, NOC and CBO are very significant, WMC somewhat significant and LCOM is insignificant in all classes.
Chidamber *et al.*, 1998	Results from three commercial object-oriented systems show that high values of CBO and LCOM are associated with lower productivity, greater rework and great design effort.
Harrison *et al.*, 1998	Investigation on CBO as a coupling measure. No relationship is found between class understandability and coupling, and limited evidence for the relationship between increased coupling and fault density.
Daly *et al.*, 1996	Study on the impact of DIT on maintainability of object-oriented software. Certain relationship is found to exist between these two measures.
Churcher and Shepperd, 1995	WMC is criticized for being language and assumption dependent.
Hitz and Montazeri, 1996	CBO is criticized for treating equally different types of coupling.
El Emam *et al.*, 1999	Study on possible confounding effect of size measure in numeric characterization of software systems.

Table 4: Empirical studies validating CK metrics

Chidamber *et al.* (1998) explore the applicability of CK metrics for managerial purposes on three commercial object-oriented systems. The study is aimed to examine the relationships between these metrics and cost, quality, and productivity. The results show that high values of CBO and LCOM are associated with lower productivity, greater rework and greater design effort. Also, it is shown that the suite provides additional information compared only to size measures.

Harrison *et al.* (1998) investigate if CBO is a good measure of coupling and whether it can be a good indicator of class understandability and fault density. The study analyzes two coupling measures: CBO and Number of Associations (NAS) across five software systems. The results indicate that there is a strong relationship between these two measures, implying that only one is needed for the assessment of a system. No relationship is found between class understandability and coupling, and only limited evidence link increased coupling and fault density.

Daly *et al.* (1996) investigate the effects of DIT on the maintainability of object-oriented software. The results suggest that software systems with the level three of inheritance show reduced time required for maintenance compared to the systems with no inheritance. However, the systems with the inheritance level five require more effort with the same purpose.

Some authors have also questioned the correctness and the usefulness of this metrics suite. WMC is criticized for being language and assumption dependent (Churcher and Shepperd, 1995); CBO for naively treating all kinds of coupling as equal (Hitz and Montazeri, 1996); LCOM for being counter-intuitive (Hitz and Montazeri, 1996). There are several redefinitions of LCOM to take into account cohesion stemming from a method invoking another method, and other aspects. El Emam *et al.* (1999) claim that the empirical usability of most of the metrics in the suite comes from the strong relationship of these metrics with the size measure.

Some researchers also suggest that these metrics can be supplemented with some other object-oriented measures, to provide a more complete picture (Li and Henry, 1993; Tang *et al.*, 1998; Briand *et al.*, 1999a).

A number of alternative object-oriented measures have been proposed. Some of them account for deficiencies of the CK suite (Li, 1998). The metric suite proposed by Li (1998) consists of the Number of Ancestor Classes (NAC), Number of Local Methods (NLM), Class method complexity (CMC), Number of descendent classes (NDC), Coupling through abstract data type (CTA), and Coupling through message passing (CTM).

Marchesi (1998) introduces metrics for object oriented analysis models in UML. Nesi and Querci (1998) propose a set of complexity and size metrics for effort evaluation and prediction, providing also a validation for some of them. Reyes and Carver (1998) define an object-oriented inter-application reuse measure. Shih *et al.* (1998) propose a concept of unit repeated inheritance and inheritance level technique for measuring software complexity of an inheritance hierarchy. Bansiya and Davis (1999) introduce Average Method Complexity (AMC) and Class Design Entropy (CDE) that measure the complexity of a class using the information content. Kamiya *et al.* (1999) propose revised set of CK metrics for software with reused components. Miller *et al.* (1999) propose four new measures of hierarchy, inheritance, identity, polymorphism, and encapsulation in an object-oriented design. Teologlou (1999) describes the predictive object points for size and effort estimation.

Despite the criticism, CK metrics have still been widely cited and adopted since they are simple and intuitive to use and, as proved in many experimental studies, they have shown the usefulness to construct prediction systems for size and number of defects. The additional convenience of the CK metrics lies in the fact that several automatic-modeling tools, such as Rational Rose, GDPro, and Together are able to compute them.

2.2 Software metrics scales

Whatever measures are chosen to model software systems behavior, it is essential to build correct prediction systems. In this sense, the notion of measurement scale is essential for understanding possible restrictions and applicability of different types of analyses for the available data.

Measurement scale is defined as the pair of the measurement mapping and the set of empirical and numerical systems. This mapping from one acceptable measure to another is called an admissible transformation (Fenton and Pfleeger, 1997). The more restrictive the class of admissible transformations, the more sophisticated the measurement scale. For example, the class of admissible transformations for measuring length is very restrictive.

All admissible transformations are of the form:

$$M' = aM$$

where M is the original measure, M' is the new one, and a is a constant.

Nominal scale is the most primitive form of measurement. There is no ordering among the classes defined by a nominal scale. Any distinct symbolic representation represents an acceptable measure on a nominal scale.

Ordinal scale contains information about the ordering of different categories, which does not have to be numeric. This ordering is based on the empirical attributes. Any mapping that preserves the ordering is an acceptable transformation. This scale contains only ranking information, so operations such as addition, subtraction, and other arithmetic operations are not possible.

Interval scale is more sophisticated and carries more information than nominal and ordinal scales. This scale captures information about the distance between different categories. An interval scale preserves differences but not ratios. Addition and subtraction are acceptable, but not multiplication and division.

Ratio scale is common in physical sciences and engineering. This measurement mapping preserves ordering, intervals between entities, and their ratios. There is a natural zero measure, representing lack of the measured attribute. All arithmetic operations can be meaningfully applied to this scale.

As an example, size and length measures are all ratio measures. Consequently, software size expressed in LOC can be considered a ratio scale. Clearly, it is possible to have a software module with zero LOC. A ratio of the size of two programs can also be calculated.

Absolute scale is the most restrictive scale with respect to the set of admissible transformations. This measure represents counts of empirical entities. All arithmetic operations on such counts are possible. Absolute scale is typical in software engineering. Number of defects for a class is an example of a measure on an absolute scale. All the metrics from the CK suite are also absolute.

Understanding scale types is essential in determining what type of statistical analysis is applicable to the given data. For example, it is inappropriate to compute ratios with any scale below ratio scale. Common problems include treating nominal and ordinal scale data as if they were numeric data (Kitchenham *et al.*, 2001). A part of this study addresses the issue of software measures assuming narrow ranges on absolute scale.

Table 5 presents a summary of the meaningful statistics for different scales types. Every meaningful statistic for lower scale type is also meaningful for a higher one.

Scale	Admissible transformations	Software engineering	Relations	Statistics	Applicable methods
Nominal	1-1 mapping	Types of defects	Equivalence	Mode, frequency	Non-parametric (e.g. Spearman correlation)
Ordinal	Any monotonically increasing function from M to M'	Severity of defects	Equivalence, greater than	Median, percent	
Interval	Linear increasing function $M'=aM+b$ ($a>0$)	Timing of defect occurrence	Equivalence, greater than, ratio of intervals	Mean, variance	
Ratio	Linear increasing function passing through 0 $M'=aM$ ($a>0$)	Program size	Equivalence, greater than, ratio of intervals, ratio of values	Geom. mean	Non-parametric and parametric (e.g. Pearson correlation)
Absolute	Identity $M'=M$	Defect count			

Table 5: Software metrics scale types
(taken in part and with modifications from Fenton and Pfleeger, 1997)

The logic behind the above proscriptions for use of statistical methods is that statistical measures should remain invariant under the admissible transformations for a particular scale (Briand *et al.*, 1996). There are two types of invariance. First, invariance in value, where the numerical value of the statistic remains unchanged under the admissible transformations. Second, invariance in reference, where the value of the statistic may change, but it would still refer to the same item or location. For example, the value of the median may change but it would still refer to the item at the middle of the distribution under monotonic increasing transformations. The item at the mean would remain the same under linear transformations even though the value of the mean changes.

Since a majority of software metrics used in this research assume values on absolute scale, and some of them recent studies evidence narrow ranges, this study addresses the issue of absolute measures with narrow ranges.

2.2.1 Analyzing software measures assuming absolute values in narrow ranges

When measures are on an absolute scale, they assume only positive integers. If the range of such integers is limited, extra care should be paid in the analysis, as the "usual" parametric statistical analysis techniques do not work properly (Fenton and Ohlsson, 2000).

The explanation of this problem with an example is provided. Suppose that and attribute *A* is measured on an absolute scale and the obtained results are as presented in Table 6.

Value of the measure of attribute A	Frequency of the measure
0	20
1	10
2	5
3	3
4	2

Table 6: Example of collected measures on absolute scale

Using the "usual" parametric methods it could be said that the data have a central tendency (represented by its mean) of 0.925 and a spread (represented by its standard deviation) of 1.186. Moreover, the 95% confidence interval computed over a normal curve is [-1.40,3.25].

Clearly such numbers would make little sense. A measure on an absolute scale would never assume a value of 0.925, nor its confidence interval includes a negative number. In addition, the range [1.40, 3.25] has size 4.65, while it is known that all data is in the interval [0,4] of size 4 and 95% of it is in the interval [0,3] of size 3.

Such measures are not appropriate for use in statistical models since they do not have enough variance and thus are likely to produce incorrect results.

In cases like this, it is more appropriate to apply other statistical measures such as median, mode and range, instead of mean and standard deviation.

To determine the size of the ranges of the metrics, a "small range" is defined for the used software measures. In this research, a range of size 10 is considered small, in the sense that data spanning over only 10 points cannot be considered normally distributed. As the target metrics have as lower bound the value 0 in the vast majority of cases, a threshold value of 10 is set for the analysis. In order to address this issue, nonparametric statistical sign test is applied to these measures.

3. Data description

This research applies several techniques in order to describe and model data on software projects and products. Different data sets are used in the experiments depending on the experimental goals and the nature of available data. This section describes the data used in this study and the collection process, where applicable.

3.1 Public domain data

One of the data sets used in this study, is a set of public domain projects downloaded from the web. 100 Java and 100 C++ software projects of different sizes are downloaded and Lines of Code and CK metrics are extracted from the code using WebMetrics, a java-based tool for metrics extraction (Succi *et al.*, 1998).

A summary of the class sizes for the 100 Java and the 100 C++ projects is presented in Table 7.

	Min	Max	Median	St. dev.
Java	28	936	83.5	169.58
C++	30	2520	59	268.29

Table 7: Descriptive statistics for the number of classes in the 100 C++ and 100 Java projects

A major disadvantage of public domain data sets is the lack of their external attributes, such as number of defects or modifications. Since external attributes are essential in expressing the behavior of software products, this is an obvious limitation. Therefore, all the analyses based on these data are focused only on internal attributes of underlying software, in particular the problem of collinearity between some CK metrics and the notion of software metrics scales.

A further drawback of the public domain data sets is the lack of sufficient information on the projects, since they do not come from controlled experiments. Only metrics extracted from the code were available, and detailed descriptions are lacking for most of them.

This is a common problem (Cohen, 1977; Cook *et al.* 1994; Liao 1998; Belanger, 1997) and therefore proper validation of the methods applied to this data is necessary.

3.2 Industrial data from telecommunication domain

Apart from the publicly available data sets, this study also analyzes industrial data. Analyses based on industrial data are more valid for the following reasons:

1. More control is possible over the experiments
2. Measures of external attributes are usually available
3. The results are more likely to be applicable on real-world systems

The available projects were written in C++, and the external measure used in devising models is the number of revisions for each C++ file. While it is true that this number includes both revisions for the purpose of defect fixing and those for enhancement reasons, this is the most accurate information that was available.

The industrial data used in this study comes from a North American telecommunication company. Again, software metrics, including LOC and CK metrics, have been extracted from the code using WebMetrics tool from two consecutive releases. The first release contains five data sets, while the second release contains seven applications. The second release also contains the shared part of code (such as libraries) that is used by all applications. Since this part, named Common Software, is larger than all other applications, its inclusion in analyses can introduce bias. Therefore, Common Software is treated as a separate application, and its inclusion is examined carefully.

Descriptive statistics of size in number of classes and LOC for each project are provided in Table 8.

The CK metrics are class-level metrics, and the external metric available for this study is a file-level metric. Therefore, a major issue was how to treat those files containing more than one class.

Several approaches were considered in order to tackle this problem:

- Assign each class the number of revisions for the file it belongs to (clearly inflates the number of revisions)
- Divide the number of revisions by the number of classes in the actual file, and assign them to classes equally (number of revisions is not an integer any longer

and it is not assigned correctly to classes in all cases except when two classes are contained in a file)

- Divide the number of revisions by the number of classes in the actual file and assign to classes according to the class size (number of revisions is not an integer any longer)
- Treat only files containing one class

Having examined all the cases mentioned above, it was shown that all of them introduce some bias into the data. Finally, the last approach was chosen, since it is the cleanest one for further analyses.

		Number of classes	**LOC/proj**
Release 1	Project 1	93	8802
	Project 2	120	6496
	Project 3	101	25325
	Project 4	38	17791
	Project 5	44	5316
Release 2	Project 1	247	19238
	Project 2	21	2212
	Project 3	71	9646
	Project 4	215	8330
	Project 5	33	1388
	Project 6	16	287
	Project 7	68	1064
	Common Software	575	30921

Table 8: Descriptive statistics for the number of classes and LOC across projects in two releases of industrial telecommunication data sets

In conclusion, this study considers only those files containing one class, as a representative sample of the system. Since the data on defects were not available at the time of study, the results are based on the number of modifications as the closest approximation for the number of defects.

For the purpose of applying Seemingly Unrelated Regression method to generalize regression models across projects, the time dimension of the data was also needed. For this purpose, the date of the latest revision for each file is used.

3.3 NASA Ames data set

The NASA Ames Research Center, as part of a program designed to assess the usefulness of Verification and Validation tools for avionics systems, collected the data on software errors and faults in the early 1980s.

The objective of this data collection effort was to discover the distribution of errors by category. The data is organized in a single file of the following description:

1. Project Identification - a company code and project code to uniquely identify each record in the file
2. Software Problem Report (SPR) Number – a tree digit number that identifies each SPR
3. Dates of filing, analyzing and completing software problem reports
4. Error Category for each problem report and when in software development process it occurred
5. Difficulty of correction
6. Number of Modules Changed by each SPR

This study uses the following variables to analyze software problem reports:

Independent variables:

- Origin of SPR
- Error Category
- Number of Modules to Change

Dependant variable:

- Difficulty of Correction

All the variables except Number of Modules To Change are discrete and can assume only specified values from the predefined sets of values, and all of them are uniformly distributed. The breakdown of possible values for the chosen attributes is shown in Table 9.

Description	Type	Possible values
Origin of SPR	Integer	0 – Revision Request
		1 – Requirements Problem
		2 – Design
		3 – Coding
		4 – Software Not in Problem
		5 – Reported in previous SPR
		6 – Other
		7 – Unknown
		8 – Correct Error from Previous SPR
		9 – Fix Comments in Source Code
Error Category	Integer	0 – Software Not in Error
		1 – Computation
		2 – Logic
		3 – Data I/O
		4 – S/W Interface
		5 – Data Handling
		6 – Database
		7 – Other
		8 – Improper Initialization
		9 – Hardware Logic
		A – Data Output
		B – Data Definition
Difficulty of Correction	Integer	0 – Info not provided
		1 – Less than 1 hour
		2 – 1 hour to 1 Day
		3 – More Than 1 Day

Table 9: Description of NASA Ames avionics data set

The dependent variable, difficulty of correction, is modified for this study. The reasons for this are as follows:

- Only few data points have value 3, which practically leads to ignoring this value
- Value 0 does not have practical value for this study, since it can be any of the remaining values

Therefore, all the records with the value 0 for difficulty of correction are removed from the data set.

The initial data set contains 3,700 records. However, in some parts it tends to be incomplete, i.e. some information may not have been filled out. Hence, incomplete records have been removed from the data set, resulting in 2,430 records to be analyzed. Finally, categories 2 and 3 are merged thus creating binary output.

NASA Ames data set is used by this study to model the effort required to fix the problem reported in SPR using decision trees. The results and a discussion are provided further in the study.

3.4 Simulated data sets

For the purpose of experimenting and comparing the proposed methods, this study also uses simulated data sets.

Since the software data are commonly non-normal in distribution and include a considerable portion of outliers, it is difficult to distinguish the consequences of each separate characteristic. Generating data sets allows researchers to control the characteristics of such data sets in order to systematically explore the relationship between accuracy, choice of prediction system and data set characteristics.

Simulated data have been used with this idea in the related studies (Shepperd and Kaddoda, 2000; Pickard *et al.*, 1999). The additional controllability is possible since only selected properties can be varied for different experiments, which clearly is not feasible with the real data. In addition, one of the common problems with software data sets, small size, can be alleviated in this way; data sets of arbitrary sizes can be generated.

Pickard *et al.* (1999) generate data sets step by step starting from the normally distributed data. Then, the independent variables are skewed, and unstable variance and outliers are introduced.

Shepperd and Kaddoda (2000) also start from the normal distribution and introduce outliers and multicollinearity. In addition, this study generates both small and large data sets, in order to investigate if certain models are better for each of them.

Both studies treat all the combination of non-normal characteristics, attempting to identify regression method most appropriate for particular data set.

This study also uses simulated data to test the devised models. Artificial data sets are generated so that the dimensionality of the data is preserved (six CK metrics and the number of revisions). Two types of data sets are produced.

First, all measures are distributed uniformly. Second, the data is additionally modified to reflect the observed behavior of the real data sets (low values of two measures, skewed distribution with some outliers etc.).

The results obtained in this way are compared with those based on the industrial data.

3.5 Software inspection data set

In order to investigate issues on software inspections a separate data set is used. This data set comes from the Center for Empirically Based Software Engineering (CeBASE, http://www.cebase.org/).

CeBASE was organized to support software organizations in answering the key questions about software processes. The goal of this organization is to accumulate empirical models in order to provide validated guidelines for selecting techniques and models, recommend areas for research, and support software engineering education.

A major area of interest of this organization is experimentation on the software inspection process. In that sense, collecting experimental data on different aspects of this process is crucial.

Eight projects on software inspections have been conducted in different environments. Two projects were undertaken in industrial environments and six of them were a part of the undergraduate courses in software engineering taught at the University of Southern California and University of Maryland.

A major goal is to collect raw data from all mentioned experiments and store them in the identical format, so that different researchers can combine and further analyze them in different ways.

However, since collecting software data is always a difficult and lengthy procedure, at the time of this research only limited data was available. Therefore, the scope of the study based on this data is also limited.

In this analysis only two data sets are used and they are described here.

Each group is given four software projects to review. They are:

1. ATM Machine (ATM), requirements document
2. Parking Garage Control System (PGCS or PG), inspection of requirements document, design, and code
3. ABC Video Store (ABC), requirements document
4. Loan Arranger (LA), requirements document and design

Since each team inspected all four applications, it can be said that the total of eight data sets is available for analysis.

Attribute	Description
Group	Number is assigned to each team consisting of 3 or 2 members who are conducting a review
Technique	Reading technique used to review the documents
Time	Length of the inspection procedure
Defects found	Actual number of defects (Number of defects is known for each project since they were "seeded")
% of Defects found	Percentage of found defects

Table 10: Description of available software inspection data

This study uses meta-analysis of effect sizes to compare and combine the results of effectiveness of different reading techniques. In particular, this meta-analysis focuses on the question whether other, so-called scenario-based reading techniques are more effective in defect detection than ad hoc technique.

3.6 Data collection process

The tool used for collecting the public domain and industrial data in this study is WebMetrics, a research system for software metrics collection (Succi *et al.*, 1998). The main feature of the system is the variety of metrics extraction tools available for different languages – C, C++, Java, Smalltalk, and Rational Rose petal format. The WebMetrics tool is written in Java.

Since the field of software metrics is constantly changing, there is no standard set of metrics, and new measures are always being proposed. Therefore, metrics researchers have to modify their existing parser tools in order to accommodate the new measures. This is a real challenge since such tools usually have very complex parser-generator and language-semantics related source code. It is also easy for metrics researchers to inject errors while modifying the large amounts of code involved. Therefore, it would be desirable to decouple the information extraction process from the use of the information. More specifically, the language parsing should be decoupled from the metrics analysis portion of the process. This requires an additional layer of abstraction with an associated intermediate representation, as shown in Figure 4.

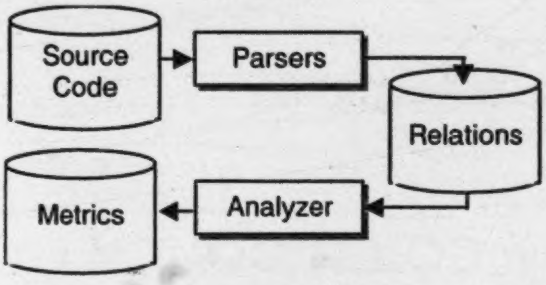

Figure 4: The process of extracting and analyzing software metrics using WebMetrics

The tool parses source files into relation files. The relations describe the existence and relations between entities found in the source files. Then, the tool analyzes all the relations and calculates metrics from them.

The relations produced by the metrics tools conform to the WebMetrics Relations Language (WRL) – a high-level, metrics-oriented intermediate representation used to convey the structure of a source program. The structure of a system is based on entities (such as classes and functions) and their interactions with each other. WRL describes a set of such relations. All of the WebMetrics language parsers output WRL.

Each metrics tool consists of a grammar parser, a symbol table, and supporting classes. The grammar parser recognizes the syntax of a particular language and is written in JavaCC (Metamata, 2001). The metrics tool is written in Java 2, and it can work on all platforms with an adequate virtual machine.

The extra layer of abstraction inserted into the metrics analysis process creates a more modular architecture overall. There are tradeoffs involving this approach, but the benefits seem to outweigh the shortcomings.

Metrics researchers only need to deal with the high-level, metrics-oriented intermediate representation when adding or modifying metrics to calculate. This spares them from having to know intimidating details about how a language is parsed. What this means in the end is that modifications can be done more easily, more quickly, and with less chance of injecting errors into the existing source code.

In addition, the breakdown of the metrics extraction process into modules offers more opportunities for reuse. Each module and abstraction layer is a point of reference for reuse by other modules.

On the other hand, adding an extra layer of abstraction means that initial development time will be longer, since the developer needs to spend more effort in the design of the modules and the intermediate representation. However, the savings in maintenance effort later on in the development lifecycle offset this disadvantage. Performance will likely degrade with the extra layer, but that is expected when providing more flexibility.

The relations are designed in form of a logic language, as Prolog-like clauses. This structure is ideal for describing language-entity relations. Currently, the following relations are defined (Table 11). They have been chosen to specifically facilitate the calculation of certain OO design and procedural metrics:

Relation	**Description**	**Simple Example**
hasLOC(entity, *x*)	The specified entity has *x* lines of code.	hasLOC(Stack, 6)
hasClass(entity, class)	The specified entity contains the specified (inner) class.	hasClass(Stack, Stack::Iterator)
hasMethod(entity, method)	The specified entity has the specified method.	hasMethod(Stack, Stack::push)
hasAttribute(entity, attribute, typename)	The specified entity has an attribute of the specified type.	hasAttribute(Stack, Stack::size, int)
hasMetric(entity, metric, value)	The specified entity has the specified value for a particular metric.	hasMetric(A.aMethod#0#, FanIn, 0).
hasFile(filename).	A parsed entity includes the specified file.	hasFile(D:/Parsers/Include/VC/include/winver.h).
extends(entity, class)	The specified entity is a specialization of the specified class.	extends(Stack, Collection)
calls(entity, method, *x*)	The specified entity called the specified method *x* times.	calls(Stack::push, Stack::isFull, 1)
usesAttribute(entity, attribute, *x*)	The specified entity uses the specified attribute *x* times.	usesAttribute(Stack::isFull, Stack::size, 2)

Table 11: Set of WebMetrics relations

The metrics tool calculates a predetermined set of metrics shown in Table 12. New metrics can be programmed using the provided API.

Object-Oriented Metrics	Procedural Metrics	Reuse Metrics
Class LOC NOM DIT NOC CBO RFC LCOM	Function LOC McCabe's cyclomatic complexity Halstead volume Information flow (Fan-In/Fan-Out)	Internal and external reuse level, frequency, and density for classes, objects, and files

Table 12: Summary of the metrics currently possible to extract using Web Metrics

The metrics for NASA Ames data sets were adopted as provided, and no data collection process was necessary. Also, the data sets for software inspection meta-analysis were readily available. Simulated data sets were generated using procedures explained before.

4 Modeling software data using software metrics

Software engineering is based on questions about the process of software development and different sets of data collected from the process. The goal of software engineering is to understand the data and answer the questions in the best possible way.

In general, a model is an abstraction of reality. It makes it possible to view the entity or concept from a particular perspective by removing less significant details. Models are used for descriptive purposes, to explain relationships between the different components of the system. They can be formulated in different forms: as equations, mappings, diagrams etc.

In addition to their descriptive applications, models are also used for prediction purposes. Based on the baseline models developed on the historical data, it is possible to make reasonable predictions of the future behavior of the analyzed system.

This section outlines the most commonly used models in software engineering and describes in details those applied in this research.

4.1 Statistical models

A widely accepted approach of modeling software data based on software measures is by applying regression models. This approach is used for a wide variety of purposes and there are many examples of studies confirming the applicability of different regression models depending on the research goals and the nature of the underlying data.

This section outlines the common characteristics of software data, and typical challenges that software engineering researchers face when attempting to build models based on them. In addition, the most widely applied statistical models are outlined followed by the description of their advantages and shortcomings. These features are important to understand, since one of the goals of this research is to reach beyond the scope of individual models and create unique models across different projects.

In order to construct and validate software engineering models it is necessary to collect the data, which is a time consuming and difficult process. In the current software engineering practice, one of the major problems researchers face is the availability of appropriate data of consistent and high quality. In particular, it is difficult to ensure that the data collected is accurate, consistent and complete (Shepperd and Cartwright, 2001). Software engineering data sets often have a number of characteristics that make analysis difficult (Gray and MacDonell, 1997). These difficulties include missing or sparse data, strong collinearity between the variables, complex non-linear relationships, unbalanced data, outliers, and the impossibility to generalize the obtained models.

A comprehensive study by Strike *et al.* (2001) evaluates techniques for prediction based on data sets with missing data points in cost estimation models. Ten different techniques for dealing with missing data are assessed on the total of 825 simulated study points. The authors conclude by offering practical and substantiated guidelines for researchers and practitioners constructing cost estimation models when their data sets have missing values.

The problem of building prediction models when data is sparse is addressed in Shepperd and Cartwright, (2001). The sparse data method described in this paper is based upon a multi-criteria decision-making technique known as *Analytic Hierarchy Process* (AHP), which represents the problem hierarchically by decomposing it into smaller, more meaningful chunks. The results are based on two industrial data sets, showing that the proposed method can be applied successfully for effort prediction.

Another distinct area of concern is the acceptability and validation of the models. This includes the issue of the model explaining its predictions. Without sufficient semantic meaning attached to the model, a satisfactory level of validation is unlikely to be achieved. This problem is made even more serious by the small data sets commonly used for building the models, which sometimes produce counterintuitive results (Shepperd and Kaddoda, 2000).

Unbalanced data is another problem in modeling software data. A balanced data set should have equal number of available data points for each combination of the values of the independent variables. Clearly, such data sets are extremely rare in empirical data coming from the software industry. The two major problems caused by lack of balance are that the impact of factors can be concealed and that spurious correlations can be observed.

Kitchenham (1998) proposes a procedure for analyzing unbalanced data sets. This method is based on the forward pass residual analysis, similar to stepwise regression, to identify the most significant factors. The procedure is demonstrated on two simple artificial data sets with only three ordinal-scale independent variables with three levels each. Although potentially useful method for specific data sets, this approach is far from being general.

The final area of concern is generalizability. Since the first models based on software metrics were derived, attempts have been made to apply the models associated with them to other projects within the organization, or even to other organizations. Use of standard COCOMO coefficients in cost estimation is an example of such an attempt. The need to recalibrate a model for a new environment has been recognized and supported by numerous authors (Kemerer, 1987). Even the models that are easily regenerated, such as linear regression models, have problems with generalizability, given their susceptibility to outliers.

Some of the issues mentioned here, collinearity and low values of some metrics are addressed in this work in order to investigate this phenomenon and suggest possible solutions in such situations.

An awareness of possible approaches helps assure that the most appropriate model is developed through employing the most suitable alternative. In some cases, the combination of the methods may be useful, each providing estimates that can be combined.

Frequently, relatively straightforward methods, such as linear regression procedures, are used to develop simple, but useful, prediction systems (Cartwright and Shepperd, 2000; Succi *et al.*, 2001; Briand and Wüst, 2001). Linear regression model is a popular method in software metrics studies and it is also used in many different ways, often in combination with various transformations to permit non-linearity.

The linear regression model can be written as:

$$y = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n + \varepsilon$$

where y is the dependent variable, the x_i's are independent variables, and ε is the error. The β_1 through β_n are the parameters that indicate the effect of a given x on y. β_0 is the intercept that indicates the expected value of y when all of the x's are 0.

A number of assumptions are made to complete the specification of the model (Long, 1997):

1. Linearity is the assumption that the dependent variable is linearly related to the independent variables used in the model. Nonlinear relations in the model are possible through the inclusion of transformed variables.
2. Collinearity is the assumption stating that the independent variables x_i used in the model have to be linearly independent. This means that none of the metrics can be represented as a linear combination of the remaining predictors in the model.

A second set of assumptions concerns the distribution of the error.

3. Error ε can be thought of as an intrinsically random, unobservable influence on the dependent variable. Alternatively, ε can be viewed as the effect of a large number of variables excluded from the model that individually have small effect on the dependent variable.
4. The assumption of the zero conditional mean requires that the conditional expectation of the error is equal to zero, implying that, for a given set of values for the independent variables, the error is expected to be zero.
5. Furthermore, the errors are assumed to be homoscedastic and uncorrelated. Homoscedasticity represents the constant variance of error, independent on x's. The errors are also assumed to be uncorrelated across different observations.
6. Finally, it is assumed that errors, as a combination of many small, unobserved factors, are normally distributed (Long, 1997).

It is often the case that most of the assumptions of a linear regression model are not satisfied for the empirical software engineering data. However, this model is convenient due its simplicity, and is often used to perform initial investigation of certain phenomena.

Other models proposed and used with software data are: Poisson log-linear model, Negative Binomial and Zero-inflated Negative Binomial Model.

The Poisson process is a simple model for occurrence of random variables that assumes the probability of an arrival in a small interval determined by the independent variables is determined only by the size of the interval, not on the history of the process to that time (Papoulis, 1991). A Poisson distribution is the distribution of the numbers of events resulting from a Poisson process.

The Poisson distribution for a dependent variable *y*, and a vector of *n* independent variables $\boldsymbol{x}=(x_1, \ldots x_n)$ is given by:

$$\Pr(y \mid x) = \frac{e^{-\mu} \cdot \mu^{y}}{y!}$$

where μ is the mean value of the dependent variable.

The Poisson distribution requires equidispersion of the data, that is, the conditional mean and the conditional variance of the dependent variable should be equal (Briand and Wüst, 1999b):

$$E\,(y|\boldsymbol{x}) = \mathrm{Var}(y|\,\boldsymbol{x})$$

However, empirical software data are often over-dispersed, that is, the value of the conditional variance is higher than the conditional mean of the dependent variable in the Poisson regression model. The main reason for this is the lack of complete control over experiments, attributing a great part of the variability in the observed data to unknown sources. This is also known as unexplained heterogeneity.

An extension of the Poisson regression model, the Negative Binomial Regression Model, allows the conditional variance of the dependent variable to exceed the conditional mean.

The Negative Binomial Regression Model can be derived from the Poisson distribution based on the unobserved heterogeneity by accounting for the combined effect of unobserved variables omitted from the original model (Gourieroux *et al.*, 1984).

The Negative binomial distribution corrects three main sources of poor fit that are often found when the Poisson distribution is used:

- Variance of the Negative binomial-distributed dependent variable exceeds the corresponding variance of the Poisson distribution for the given mean.
- Increased variance in the Negative binomial results in substantially larger probabilities for small counts.

- Probabilities for larger counts are slightly larger in the Negative binomial distribution.

The resulting Negative Binomial Regression Model is the most commonly used model based on the combination of the Poisson distribution with other distributions (Long, 1999).

Furthermore, Zero-inflated models allow the possibility that different processes generate zero counts and positive counts. They assume that two different groups form the population exist. An item belongs to one of the groups with probability ψ and to the other with probability $1\text{-}\psi$. This probability is determined from the characteristics of the item (Lambert, 1992; Greene, 2000). Items in the first group always have zero counts. Such items are different from those that have zero counts with a certain probability. The latter belong to the second group together with the items with non-zero counts. This approach improves the performance of models where a significant part of dependent variable has zero value.

Finally, a method combining regression modeling and classification algorithms is found in Logistic regression (Hosmer and Lemeshow, 1989; Khoshgoftaar, 1997). The assumption of this model is that the dependent variable is of binary nature.

A multivariate logistic regression model is based on the following equation:

$$\pi(x_1, x_2, \ldots, x_n) = \frac{e^{(C_0+C_1x_1+\ldots+C_nx_n)}}{1+e^{(C_0+C_1x_1+\ldots+C_nx_n)}}$$

where π is the conditional probability of obtaining one of the two possible values for the dependent variable, i.e. the output if of binary nature. Therefore, unlike with other regression techniques, such as linear or Poisson regression, the dependent variable is not measured directly. Software engineering studies based on software metrics also use logistic regression.

Briand *et al.* (2000a), Tang *et al.* (1998), El Emam *et al.* (2001) and Basili *et al.* (1996) use logistic regression to explore the relationships between design measures and software quality in object-oriented systems. The dependent variable in this study is the fault-proneness of object-oriented classes. Therefore, π is the probability that a fault is found in the class.

As previously mentioned, the nature of software data can be such that the extracted measures are not linearly independent. The following section describes the meaning and importance of this phenomenon and attempts to identify such a behavior in software data.

4.1.1 Presence of collinearity in multivariate regression models

Collinearity is the presence of a strong linear relationship between two or more independent variables in a multivariate model. Its presence usually prevents a precise determination of the statistical parameters under study (Belsley, 1991). In this subsection the explanation why collinearity is dangerous when building multivariate linear models is provided.

Suppose that the goal is the estimation of a dependant variable y using two independent variables, x_1 and x_2 (1). It is common in this case to use OLS multilinear regression and to determine the coefficients β_0, β_1 and β_2 so that the expected values of y can be expressed as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \qquad (1)$$

If x_1 and x_2 are linearly related, x_1 can be estimated with x_2 using the following linear equation, where d and f are the mean square coefficients:

$$x_1 = dx_2 + f \qquad (2)$$

(1) can be restructured as follows:

$$y = \beta_1 x_1 + \beta_2 x_2 + tx_1 - tx_1 + \beta_0 = (\beta_1 + t)\, x_1 + \beta_2 x_2 - tx_1 + \beta_0 \qquad (3)$$

where $-t$ is a real number.

Then the term tx_1 can be substituted with $t(dx_2+f)$:

$$y = (\beta_1 + t)\, x_1 + (\beta_2 - td)\, x_2 + \beta_0 - tf \qquad (4)$$

And since there are infinitely many real values of t for which $(\beta_1+t) \neq \beta_1$ or $(\beta_2-td) \neq \beta_2$ or $(\beta_0-tf) \neq \beta_0$, the expression (1) is not unique. This means that a solid, unique model cannot be built, since the effects of the independent variables are unclear: even the direction of the effect can be changed, as nicely evidenced by (Fenton and Neil, 1999).

Statistics textbooks warn to check for the presence of collinearity and suggest that, if collinearity appears, one of the independent variables be discharged (Aron and Aron, 1997).

Belsley (1991) offers a comprehensive analysis of collinearity, and provides a review of and suggestions for possible diagnostic and improvement techniques. Among them are:

- Examining the eigenvalues and eigenvectors of the cross-product matrix.
- Studying the correlation matrix of the least square estimator based on partial correlations between any two columns in this matrix.
- Using the correlation matrix and its inverse, under the assumption of a normal data distribution.
- Investigating correlation matrix of explanatory variables since a high correlation implies collinearity.

Recent studies also confirm the existence of collinearity between object-oriented software measures.

In (Chidamber *et al.*, 1998), the authors evidence the presence of collinearity between CBO, NOM, and RFC studying three industrial projects written in C++ and Objective C. The collinearity is diagnosed based on significant correlations calculated between these measures. The study includes three industrial software systems and the results indicate that such a correlation exists in general.

The work of Basili *et al.* (1996) concludes that the correlation between CBO, RFC and NOM is partly significant and quite weak. The analysis based on Pearson's correlations concludes that these measures are mostly independent in statistical terms, and do not capture redundant information excessively.

This research aims at determining if *in general* there are significantly high, linear relationships between the three measures from the CK suite, CBO, NOM, and RFC. Since the focus is on only three variables, the simple approach of studying the correlation matrix of the least square estimators is taken. Had more variables been dealt with, other techniques should have been introduced and the problem of mutual dependence among three or more variables should have been considered at the same time. If such a relationship is confirmed in general, in case of linear or log-linear models, all but one of collinear models should be removed from the prediction model.

The method applied to deal with this is the meta-analytic technique of Weighted Estimators of a Common Correlation. The method itself is described in detail (Section 4.3.1) and the results are provided later in the research.

As discussed previously, regression models represent an important way to model software data. However, there are occasions when it is needed to consider regression models jointly in order to achieve more general results from experimental studies in the software domain.

4.2 Systems of regression equations

This study attempts to devise a unique regression model from several individual regression models based on separate software projects. This goal is important from the point of view of managers and developers since it should give a general picture of the current state of the art in the software development process of their organization. Also, this method could give possible predictions on the behavior of the software products once they are delivered to the customers.

For the purpose of this investigation, two consecutive releases of industrial data are considered. Two methods are applied: method of Seemingly Unrelated Regression Equations (SURE) and an extension of OLS model, named Data-model method.

4.2.1 Seemingly Unrelated Regression Equations

Seemingly Unrelated Regression Equations method is widely used in econometric studies (Greene, 2000). It focuses on treating linear econometric models and devising a unique regression model from the individual models.

Some typical applications of this method in econometric studies are:

- Devising a model for investment decisions based on variables that reflect anticipated profit and replacement of the capital stock
- Cross-country comparison of economic performance over time, depending on the labor and political organization
- Obtaining the capital asset pricing model of finance for a given security

In the studies numbered here, econometrists attempt to treat the separate models jointly, since the factors ruling the market are usually common to many companies. In such cases the models can be derived totally independently; however, the disturbances in the models should be inter-related due to some common factors. Such systems of regression equations where the errors are correlated can be handled with the SURE method. In cases where errors are correlated, the OLS method is not acceptable, and SURE models should be used.

The method is based on covariance structures of errors, and the final model is obtained by assuming that the coefficient vectors in all equations are the same. The detailed description of the method is provided below.

In case of M linear models each model can be represented with:

$$\mathbf{y}_i = \mathbf{x}_i\boldsymbol{\beta}_i + \varepsilon_i.\ i=1,\dots,M \tag{1}$$

where

$$\boldsymbol{\varepsilon} = [\boldsymbol{\varepsilon}^T_1, \boldsymbol{\varepsilon}^T_2, \dots, \boldsymbol{\varepsilon}^T_M]^T \tag{2}$$

$$\mathbf{V} = \mathbf{E}[\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^T] \tag{3}$$

and $\boldsymbol{\varepsilon}$ is the vector of errors and $\boldsymbol{V}$ is the covariance matrix of errors.

With the assumption that the errors are uncorrelated within a data set, it leads to:

$$\mathbf{E}[\boldsymbol{\varepsilon}_i\boldsymbol{\varepsilon}^T_j] = \sigma_{ij}\mathbf{I}_T \tag{4}$$

$$\text{or} \qquad \mathbf{E}[\boldsymbol{\varepsilon}_i\boldsymbol{\varepsilon}^T_j] = \mathbf{V} = \begin{bmatrix} \sigma_{11}\mathbf{I} & \sigma_{12}\mathbf{I} & . & . & . & \sigma_{1M}\mathbf{I} \\ & & . & . & . & \\ & . & & & & \\ & . & & & & \\ & . & & & & \\ \sigma_{M1}\mathbf{I} & \sigma_{M2}\mathbf{I} & . & . & . & \sigma_{MM}\mathbf{I} \end{bmatrix}$$

Covariance, in its nature, describes how much pairs of chosen variables vary together. Therefore, the covariance matrix of errors contains the information on the relationship between the individual models.

The final estimator for the unique regression coefficient $\boldsymbol{\beta}$ is obtained from the following expression:

$$\boldsymbol{\beta} = [\mathbf{x}^T\mathbf{V}^{-1}\mathbf{x}]^{-1}\mathbf{x}^T\mathbf{V}^{-1}\mathbf{y} \tag{5}$$

This represents the generalized least squares (GLS) estimator for general regression models.

In econometric studies researchers typically observe the phenomenon of interest over a longer period of time, and record the observed values. Therefore, the data often come from controlled experiments, since observations do not intrude the economic processes. Therefore, in most cases the sizes of data sets considered for such analyses are the same.

However, in the case of software engineering studies, planned and controlled experiments commonly include changes in the software development routine of the participating company. Therefore, research is often based on case studies when the whole process is completed. That is one of the reasons why software engineering data sets are almost never of the same size.

Since the original SURE model accounts only for the cases with the data sets of the same size, an extension to this method is necessary in order to analyze software data. Im (1994) and Schmidt (1977) provide such an extension that accounts for different data set sizes.

The description of the proposed extension is easily explained using the example of two seemingly unrelated regressions. Assume that the analysis is done on two data sets represented with linear models as follows

$$y_1 = x_1 \beta_1 + \varepsilon_1$$

$$y_2 = x_2 \beta_2 + \varepsilon_2$$

and that there are T data points in the first data set, and T+S data points in the second. The first T data points in both equations are assumed to match in time, which is one of the requirements of the method, and there are S extra observations in the larger data set.

Defining

$$y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, x = \begin{bmatrix} x_1 & 0 \\ 0 & x_2 \end{bmatrix}$$

$$\beta = \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix}, \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \end{bmatrix}$$

the covariance matrix of disturbances becomes more complex

$$V = \begin{bmatrix} \sigma_{11} I_T & \sigma_{12} I_T & 0 \\ \sigma_{21} I_T & \sigma_{22} I_T & 0 \\ 0 & 0 & \sigma_{22} I_S \end{bmatrix}$$

The only significant difference in the extension of the original model is the modified matrix of covariance of errors. Compared to the case with equal data set sizes, the new matrix is larger for the size S.

It is clear from the presented structure of the new covariance matrix why time ordering is necessary when building the matrix. However, even though Im (1994) and Schmidt (1977) propose exact matching in time, proper order is correct enough.

From the expression of the extended covariance matrix, it is clear that even if measures are not taken at the same moments in time but are still properly ordered, coefficients σ_{11} to σ_{TT} will still be correctly calculated. If this order in time is violated, elements of this matrix will have different and incorrect values.

With the matrix V modified to suite unequal numbers of observations, the same expression for β as shown in (5) is still valid.

One of the serious limitations of SURE model is that it is primarily concerned with combining linear regression models. Since regression models based on software data are usually non-linear or in the best case log-linear, the applicability of this model is not general.

4.2.2 Data-model method

Another method used to combine regression models across many studies is *Data-model* method. The presented method is not found in the literature, yet it is the attempt of this study to extend the scope of existing methods for generalizing regression across individual projects.

The main idea of this model is to try and minimize the error of regression both on the data and the model level. While OLS and the method of Seemingly Unrelated Regression models treat the error only at the level of data, it is hoped that this approach will provide one more dimension of looking at the data.

The mathematics background required for this approach is relatively simple. Suppose that the analysis includes *N* data sets and that for each data set there is a regression model represented by vector of regression coefficients β.

Here, the error of regression is considered at two levels. The first level attempts to minimize the square error on the data, as shown in (1).

$$Q_1 = \sum_{k=1}^{N[i]} (y_k[i] - \beta'[i]x_k)^2 \qquad (1)$$

The second level is focused on minimizing the difference between the regression coefficients devised from each separate data set. Expression $\|\beta[i] - \beta[j]\|$ denotes the distance between the vectors of coefficients in the Euclidean space. Different choice for the distance function can to certain extent influence the results.

$$Q_2 = \sum_{\substack{j=1 \\ j \neq i}}^{N[i]} \|\beta[i] - \beta[j]\| \qquad (2)$$

The sum of these errors represents the total error on both levels.

$$Q = Q_1 + \gamma Q_2 \qquad (3)$$

Coefficient γ is introduced to balance the two components. In other words, depending on the data, one component can be considerably larger.

The idea is to minimize both components at the same time and produce the optimal output for the combined regression model.

This approach has two advantages over the SURE:

1. It is easily extended to non-linear model
2. Time dimension in data is not necessary

On the other hand, a major shortcoming of this method is that it can be quite difficult to balance the two components, meaning that one component can prevail in the analysis.

The results achieved using this method as well as the comparison with the other proposed models are outlined and discussed in the results section of the study.

4.3 Methods for Generalizing Results from Individual Studies

When trying to reveal new principles that govern different phenomena, scientists usually turn to conducting research activities, commonly referred to as primary studies. In this sense experimentation is an accepted approach toward scientific disciplines, including the software development community. It is a tool for validating theories, since the classical scientific method depends upon theory formation followed by experimentation and observation in order to validate, modify and improve theory.

Primary studies are commonly followed by replications of original studies, called secondary studies, conducted by someone else than the researcher who collects the data and with the purpose of validating the results of original research.

However, empirical studies of software engineering phenomena often report different results. Therefore, apart from new and improved techniques and rules involved in conducting individual studies, there is also the need for more means of making sense of the vast number of accumulated study findings. The importance of these methods lies in the fact that they provide ways of building beyond the scope of individual research designs.

Several methods have been developed with the same goal of creating a single conclusion out of many individual studies. The background description of existing methods is outlined here. The detailed description of the methods applied in this study to generalize results coming from individual studies, such as meta-analysis, is also provided. One of the goals of this study is to investigate these methods, contextualize them in the software engineering field and assess their feasibility and usefulness to create knowledge base for future studies. The most widely used methods for generalizing results from individual studies are described.

For many years, a method called narrative review or qualitative research was used, as a way of combining results across different studies.

A narrative review is the typical review article found in most journals. Individuals who are considered to be experts in a given field produce these reviews. They often use informal and subjective methods to collect and interpret information. Such reviews are appealing because they are relatively easy and quick to produce. They are also attractive to readers because they distill the views of an authority in a field in a short piece, saving the readers time and effort.

The main problem with these reviews, however, is that the reader must take them at face value, because they are impossible to replicate. This poses a real problem in an era of information overload in which increasingly complex decisions need to be made.

Although useful as a beginning of this kind of research, these rather informal reviews show the following shortcomings (Light and Pillemar, 1984):

- Traditional reviews are subjective, which means that two scientists, using the same source, can reach surprisingly different conclusions.
- Narrative reviews are scientifically unsound. This means that reviewers can get outcomes using methods inconsistent with good statistical practice.
- This method gives poor results when the number of primary studies is large.

An attempt to alleviate some of the problems present in narrative reviews is introducing systematic reviews. A systematic review, in its ideal form, is a review that includes an explicit and detailed description of how it was conducted so that any interested reader would be able to replicate it. The ideal systematic review should incorporate strategies to minimize bias and maximize precision. In its purest form, the report of a systematic review should include a clear research question, criteria for inclusion or exclusion of primary studies, the process used to identify primary studies, the methods used to assess the methodological quality of the selected studies, and the methods used to extract and summarize the results of primary trials on which the conclusions are based. Systematic reviews clearly overcome the major limitations of narrative reviews. Their main disadvantage is that they require more time and resources to prepare than do their narrative counterparts.

In order to formalize the process of literature review, which proved inefficient in case of large number of studies, a new method of vote counting was introduced.

The main goal of the new technique was to introduce quantifiable and more objective approach to summarizing the results of many independent studies.

The process of applying vote counting is as follows:

1. Create a question like "Is there an effect of a certain parameter on the observed variable?"
2. Divide the whole set of collected studies into those where the effect exists and those where it does not
3. Assign numbers to two groups of existing and non-existing effect
4. Reach the final decision based on the higher number of votes

Although this approach introduces some measures into the process of literature reviews, it has several shortcomings. The most serious is that reviewers using the voting method treat all studies alike and completely ignore the fact that studies with different sample sizes have a completely different meaning for "significant" (Hunter and Schmidt, 1995).

As an illustration of this problem, a simple example is provided and discussed. Assume that the following data in Table 13 come from five individual experimental studies.

From the sample data and according to the described procedure of vote counting, it seems straightforward to reach a conclusion. Out of five included studies, four show significant results in answering the question about the presence of the effect. Three of these four claim that an effect exists, and one that it does not exist. The overall result seems to indicate that an effect actually exists.

Study No.	Study size	Significance level	Answer
1	12	0.01*	Yes
2	203	0.06	No
3	24	0.02*	Yes
4	347	0.05*	No
5	31	0.03*	Yes
Total size	617	Result of vote counting	Yes

Table 13: Generalizing results from individual studies using vote counting method - example

However, if the data is examined more closely, the effect exists in 67 out of 617 data points, which constitutes only 10.86% of the available data. It is questionable whether a general conclusion should be based on such a small portion of data set only considering significance levels.

There exist efforts to improve the performance of conventional vote counting (Hedges and Olkin, 1984; Hunter and Schmidt, 1995) by trying to capture the characteristics of individual studies in more detail. However, even if the problem of different meanings of statistical significance can be resolved, a further limitation of this method is its inability to determine the level of an effect.

In conclusion, vote-counting method can be used only to assess whether the effect of an observed phenomenon exists rather than to determine the level of that effect. Compared to other methods of cumulating research findings, meta-analysis produces the most reliable results (Brooks, 1997; Pickard *et al.*, 1998).

4.3.1 Statistical meta-analysis

Meta-analysis offers a set of quantitative techniques that permit synthesizing results of many types of research, including opinion surveys, correlation studies, experimental and quasi-experimental studies, and regression analyses.

This technique refers to analysis of the results of many independent studies. It is the summary and integration of the results of the previous studies, which should serve to further our understanding of the phenomena beyond the level achieved with any single investigation. Furthermore, meta-analysis can be used to build upon existing results and focus future inquiry in terms of providing clearer directions about what the remaining research studies are needed next.

Up to now, meta-analysis has been given most attention in medical sciences. Consequently, it has achieved the best results in that field. As an example, Cook *et al.* (1994) state that patients receiving additional psychoeducational care recover more quickly, experience less postsurgical pain, have less psychological distress, and are more satisfied with the care they receive.

What makes meta-analysis in this scientific discipline so successful is the partial possibility to eliminate publication bias, which presents one of the most inevitable obstacles of this kind of study. Researchers in this field have access to large databases such as MEDLINE, PsychLIT, MEDLAS, Dissertation Abstracts, ERIC, HSTAT (Health

Services/Technology Assessment Text), HealthSTAR (Health Services, Technology, Administration, Research, bibliographic citations), HSRProj (Health Services Research Projects), DIRLINE (directory of Information Resources Online) etc. These databases form a major reference for identifying existing studies on a specific phenomenon, thus reducing the chance of publication bias.

Behavioral Sciences also offer abundant examples of research synthesis. In fact, educational and psychological research activities were the first where meta-analysis was applied. One of the typical requirements is to synthesize existing research by comparing effects of one type of treatment to another. For example, the results by Liao (1998) suggest that the effects of using hypermedia in instruction are positive when compared to the effects of traditional instruction.

Several types of research synthesis have also been undertaken in the field of ecology, including narrative review and meta-analysis. The journal ECOLOGY, 1999 vol. 80 (4), has a special section on meta-analysis and its application in this science. There are seven papers covering general aspects of meta-analysis and example analysis. A successful example worth mentioning is a meta-analysis of more than 150 model stream ecosystem studies employed in hazard assessment conducted to assess the effect of model ecosystem size on biological complexity and experimental design (Belanger, 1997).

An interesting application of meta-analysis is also found in conducting the process of data mining. Sohn (1999) finds meta-analysis a useful method for determining which classification algorithms are the most appropriate for the purpose of pattern recognition in the process of data mining.

There have also been efforts to apply meta-analysis in software engineering. Some of the studies found that the common estimate could be reached from individual studies whereas others concluded that drawing a unique conclusion was not possible.

Hu (1997) evaluates four alternative software production models, which have not been applied widely in software engineering practice. The study investigates linear, quadratic, Cobb-Douglas and translog models and compares their performance. Rather than an attempt to combine the results obtained from individual studies in terms of meta-analytical approach, this study represents a comprehensive statistical analysis and comparison of these results.

The comprehensive analysis by Pickard *et al.* (1998) targets the relationship between project effort and product size in various software companies. The investigation showed that there is a high correlation between these, and that heterogeneity of data does not influence the result to a great extent. Moreover, the analysis also includes the test of sensitivity of meta-analysis, and the attempts to achieve more consistent results through exclusion of small number of data points.

The study by Wood *et al.* (1998) performs *multi-method* as an alternative to 'single-shot' empirical studies. This approach is based on the combination of complementary empirical research methods. The underlying idea is to create more robust conclusions and increased understanding of research results. This paper demonstrates an application of the multi-method approach in an empirical investigation of object-oriented technology and results indicate that the multi-method approach offers the possibility of more reliable and generalizable results from empirical software engineering research.

Hayes (2000) focused on the investigation of efficacy of different software requirements inspection techniques. The meta-analysis included five experiments, among which one was the original and four others were replications. The paper does not give a clear opinion whether the final result can be adopted, but rather provides discussion of the problems associated with the heterogeneity present in the data sets.

Another meta-analysis with software engineering data by Miller (2000) examines several defect detection techniques. The analysis included five studies and different defect detection techniques, but the author concludes that the common estimate cannot be reached, due to the diversity of the experiments and data sets.

Two distinctly different directions exist for combining evidence from different studies almost from the very beginning of statistical meta-analysis. One approach relies on estimating correlation coefficients across studies, and is named parametric, since it assumes that p-values are uniformly distributed between zero and unity. The other approach relies on transforming the available data into effect sizes and combining them across studies.

4.3.1.1 Meta-analysis based on correlation coefficients

Correlation coefficients have been used extensively as an index of the relationship between two variables. Since the correlation coefficient is a scale-free measure of the relationship between variables, it is invariant under substitution of different but linearly equitable measures of the same construct. The correlation coefficient is therefore a natural candidate as an index of effect magnitude suitable for cumulation across studies.

Correlation measures usually have one weakness as a collinearity diagnostic; there is no obvious cutoff for how large a correlation must be to indicate collinearity. In many statistical studies, correlations of the magnitude 0.45 are considered large (Barnston, 1994). These are usually studies interested in testing the hypothesis that no correlation exists. Conversely, in the studies presuming the existence of a relation and interested in knowing if it is strong, correlations like 0.45 are considered small. This study considers as a flag of possible collinearity a presence of the correlation between two variables higher than 0.5 (Cohen, 1977).

Using correlations to determine the presence of collinearity has two major advantages in this study:

- It also works with non-parametric correlations, and so is more suited to situations where no assumption can be made on the distribution of data.
- It is amenable for the application of statistical meta-analysis; therefore the results across projects can be generalized.

In this work, the meta-analytical technique based on combining correlation coefficients from many studies, called "Weighted Estimators of a Common Correlation" (Hedges and Olkin, 1985) is used.

4.3.1.1.1.Weighted Estimators of a Common Correlation

The method of weighted correlations is applied in this study to investigate whether three of the CK metrics show linear interdependence. The experiments are undertaken on both public domain and industrial data to confirm or disconfirm this.

This technique consists of 4 steps, taken under the assumption that the correlations come all from the same underlying population:

1. Elimination of the bias in the estimation of the correlation using an unbiased estimator (the *G* estimator)
2. Normalization of the data with the Fisher *z* transformation
3. Computation of the required confidence interval for the transformed correlations, in this study the 95% confidence interval
4. Application of the inverse Fisher *z* transformation on the resulting range.
5. At the end, there is a check whether the results disprove the original assumption on homogeneity of the data. For this, an extension of the chi-square test is used.

Given below, there is the summary of these 4 steps as described in (Hedges and Olkin, 1985).

Examination of the sampling distribution reveals that the sampling distribution (*r*) is a maximum likelihood of the population distribution (ρ). However, *r* is not an unbiased estimator of ρ. The exact mean and bias of *r* are obtained only as infinite series, but an approximate means value of *r*, to order 1/n, is given by

$$E(r) \equiv \rho - \rho(1-\rho^2)/2n$$

which means that the bias of *r* as an estimator of ρ is approximately

$$\mathrm{Bias}(r) = -\rho(1-\rho^2)/2n$$

Thus the sample correlation tends to underestimate the absolute magnitude of the population correlation ρ.

The *G* transform, which is an infinite series, supplies an unbiased estimator of the correlations of the populations based on the populations of the samples. An easy to compute approximation of *G* is $\tilde{G}$:

$$\tilde{G}(r) = r + \frac{r(1-r^2)}{2(n-3)}$$

where *r* is a sample correlation coefficient. It can be either parametric or nonparametric depending on the concrete data set investigated.

The error of using $\widetilde{G}$ instead of G is negligible when the number of data points is more than ten as in this study. If the sample size is moderately large (say, over 15), the bias of the sample correlation is seldom of practical concern. Even for smaller samples, if the true correlation is close to 0 or ±1, the bias is negligible. If a study with a small sample size is used to estimate a correlation, and the true correlation is in the range of 0.4 and 0.6, then the bias can be a serious concern. The unbiased estimator here is accurate within 0.01 if $n \geq 8$ and to within 0.001 if $n \geq 18$. The derivation of the unbiased estimator is given in (Olkin and Pratt, 1958).

Still, the result of the application of G is in the range [-1,1]. To normalize the data a bilinear transformation is adopted, the Fisher z transform:

$$z(r) = \frac{1}{2}\log\frac{(1+r)}{(1-r)}$$

Given a set of transformed correlation coefficients $z_1, \ldots, z_n$, the mean transformed value $\bar{z}$ can now be computed as a weighted average of the z_is.

$$\bar{z} = \sum w_i z_i$$

where the weight (w_i) for the i^{th} experiment is computed on the basis of the size of the experiment as:

$$w_i = \frac{(n_i - 3)}{\sum_{j=1}^{k}(n_j - 3)}$$

The 95% confidence interval in the transformed space, $[z^-,z^+]$, is then determined using the “usual” rule for normal distributions:

$$z^- = \bar{z} - \frac{1.96}{\sqrt{\sum_{i=1}^{k}(n_i - 3)}}$$

$$z^+ = \bar{z} + \frac{1.96}{\sqrt{\sum_{i=1}^{k}(n_i - 3)}}$$

To determine the 95% confidence interval on the original correlation coefficient, $[r^-,r^+]$, the inverse Fisher z transform, z^{-1},on z^- and z^+ need to be computed:

$$z^{-1}(z) = \frac{(e^{2z} - 1)}{(e^{2z} + 1)}$$

At the end, the original data needs to be checked for homogeneity. To do so, a chi-square test against the null hypothesis of homogeneity is performed. Given the sample size and the power of the test, a non-rejection of the null hypothesis amounts to its acceptance.

The Q statistics used in this case is:

$$Q = \sum_{i=1}^{k} (n_i - 3)(z_i - \bar{z})^2$$

We compare it with the chi-square threshold value for *k-1* degrees of freedom. If it is lower the data set is considered homogeneous, otherwise it is not.

4.3.1.2 Meta-analysis of effect sizes

Another measure to base the meta-analysis on is effect size. Effect size is the standardized mean difference between two subsamples coming from a single experiment.

Two subsamples are commonly identified as *experimental* and *control* groups. Experimental group refers to the portion of the data where some treatment exists and control refers to where it does not exist. The goal of such a study is to discover the extent to which these two groups of data differ. As explained previously, vote counting can identify whether the difference between two groups exists; however the extent of the difference cannot be assessed.

This research applies meta-analytic method of comparing and combining effect sizes to evaluate effectiveness of different software inspection techniques. In particular, the success of two reading techniques in detecting defects are compared and combined across a number of studies.

Although effect size can be calculated from only one study, in order to generalize results across many studies, the effect sizes are combined.

Hedges and Olkin (1985) provide the procedure for combining effect sizes across studies. The procedure is outlined here.

Standardized mean difference for a single study can be expressed as

$$g = \frac{\mu^E - \mu^C}{s^*} \qquad (1)$$

where μ indicates means of the dependent variable, E indicates experimental, C control sample, and s^* pooled sample standard deviation.

However, Hedges and Olkin (1985) also report that a bias exists in this definition and the correction for this estimator is

$$d = J(N-2) \qquad (2)$$

where *J* is a constant that depends on the degrees of the freedom for the specific study, and *N* is total size of sample including experimental and control group, i.e. $N = n^E + n^C$.

m	J(m)	m	J(m)	m	J(m)	m	J(m)
2	0.5642	15	0.9490	28	0.9729	41	0.9816
3	0.7236	16	0.9523	29	0.9739	42	0.9820
4	0.7979	17	0.9551	30	0.9748	43	0.9824
5	0.8408	18	0.9577	31	0.9756	44	0.9828
6	0.8686	19	0.9599	32	0.9764	45	0.9832
7	0.8882	20	0.9619	33	0.9771	46	0.9836
8	0.9027	21	0.9638	34	0.9778	47	0.9839
9	0.9139	22	0.9655	35	0.9784	48	0.9843
10	0.9228	23	0.9670	36	0.9790	49	0.9846
11	0.9300	24	0.9684	37	0.9796	50	0.9849
12	0.9359	25	0.9699	38	0.9801		
13	0.9410	26	0.9708	39	0.9806		
14	0.9453	27	0.9719	40	0.9811		

Table 14: Exact values of the bias correction factor J(m) for effect sizes (taken from Hedges and Olkin, 1985)

Table 14 shows the values of the bias correction factor for degrees of freedom from 2 to 50 and Figure 5 illustrates this relationship.

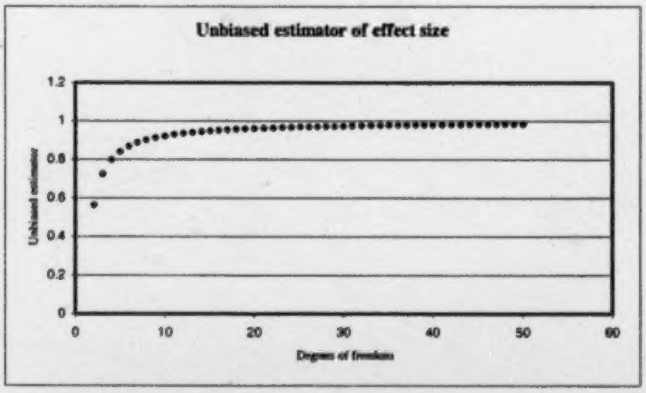

Figure 5: Unbiased estimator of effect size

The values from the table indicate that for *m* lower than 20, the bias can be substantial while for higher values of degrees of freedom it is less than 5%.

The dependence between degrees of freedom and J(m) can be approximated with the following expression:

$$J(m) = 1 - \frac{3}{4m-1} \tag{3}$$

Based on the correction factor, the unbiased estimator of effect size is as follows:

$$g = J(N\text{-}2)\,\frac{\mu^{E}-\mu^{C}}{s^{*}} \tag{4}$$

The pooled sample standard deviation depends on both subsamples, and is provided by the following expression:

$$s^{*} = \sqrt{\frac{(n^{E}-1)(s^{E})^{2}+(n^{C}-1)(s^{C})^{2}}{n^{E}+n^{C}-2}} \tag{5}$$

The described estimator refers to a single study. If a series of *k* studies can reasonably be expected to share a common effect size, the information from all the studies should be pooled.

One simple pooled estimate is the average of the estimates obtained from each study, and because of its simplicity, this method is used frequently. As might be expected, if the studies do not have a common sample size, their sizes should be taken into account in the form of weights. This becomes even clearer taking into account the fact that studies with larger sample sizes are in general more precise. Therefore, simple average can be improved by introducing weighting factors. This study uses weights based on the size of each study and is expressed in (6).

$$w_i \equiv \frac{\tilde{n}_i}{\sum_{j=1}^{k}\tilde{n}_j} \tag{6}$$

where w_i are nonnegative weights that sum up to unity, *k* is the number of studies included in the meta-analysis, and $\tilde{n}_i = n_i^{E} n_i^{C}/(n_i^{E}+n_i^{C})$.

When the weights and individual effect sizes are defined, the pooled effect size can be obtained through linear combination as shown in (7).

$$g_w \equiv w_1 g_1 + \ldots + w_k g_k \qquad (7)$$

After pooling the estimates of effect size from a series of *k* studies, it is important to determine whether the studies can be reasonably described as sharing a common effect size. A statistical test for the homogeneity of effect sizes is, formally, a test of the hypotheses that all the effect sizes of the studies are the same versus the alternative hypothesis that at least one of the effect sizes differs from the remainder.

Hedges and Olkin propose a test statistic *Q* for testing this hypothesis. This test statistic is defined in the following manner:

$$Q = \sum_{i=1}^{k} \frac{g_i^{\,2}}{\hat{\sigma}^2(g_i)} - \left(\sum_{i=1}^{k} \frac{1}{\hat{\sigma}^2(g_i)}\right)^2 / \sum_{i=1}^{k} \frac{1}{\hat{\sigma}^2(g_i)} \qquad (8)$$

With the estimated variance of d_i defined as:

$$\hat{\sigma}^2(g_i) = \frac{8 + g_i^{\,2}}{4N}$$

If all studies have the same population effect size, then the test statistic *Q* has an asymptotic chi square distribution with *k-1* degrees of freedom. If the value of *Q* exceeds the 100(1-α) – percent critical value of the chi-square with *k-1* degrees of freedom, the hypothesis that the effect sizes are homogeneous should be rejected. If the null hypothesis cannot be rejected, the effect size can be pooled in the previously described way.

The final question that needs to be addressed is whether the pooled effect size can be considered practically significant, that is, whether the effect is large enough to be of interest. This study adopts effect sizes larger than 0.5 to be of practical significance.

4.3.1.3 Controversy of meta-analytical approach

Meta-analysis is not free from controversy. Several researchers warn on risks associated with its application to software engineering (Miller, 2000; Pickard *et al.*, 1998).

1. Combining results from different experiments poses a high risk of comparing apples and oranges.
2. There is limited or no control on the quality of the data, since meta-analysts deal with results provided by other scientists.
3. It is not always taken into account whether the actual effect sizes are significant.
4. The randomization of the data sets, the prerequisite for correct meta-analysis, is sometimes not checked.
5. Scientists are not always provided with the raw data.
6. Homogeneity test is not always performed.

In this research the first problem is partially overcome by the fact that the analysis is still undertaken on code. Although the code originates in a very wide range of sources, it is still Java and C++ code.

The second problem does not arise, since the authors have full control over all the experiments. For the first meta-analysis, the data collection has been performed using a solid metrics extraction tool, WebMetrics (Succi *et al.*, 1998). In this way the measurement errors have been avoided and all the data have the same consistent quality. For the second investigation on software inspection effectiveness, the data come from the university environment where all the projects have been controlled and based on the known number of defects.

As for the magnitude of effect size, in this investigation large effect sizes for both moderator variables are reported, which casts away any doubt on the statistical significance of the results. Since there are not many related studies in software engineering dealing with effect sizes, the adopted level of significance, with respect to the existing studies in other scientific fields, is 0.05.

We also try to eliminate the bias and maintain randomization of the studies by including all the individual studies and examining detailed information for each project that was available.

Finally, the always-critical test of homogeneity is satisfied for most of the relationships of interest in this study. In the cases where the homogeneity was hardly satisfied, the results are additionally discussed.

The following issue this research investigates is the range of values two of CK metrics, DIT and NOC, assume. As explained earlier, variables that assume values in narrow ranges should be handled with care. To examine this, statistical sign test is used as follows.

4.3.2 Sign test

In order to confirm or disprove or refute the hypothesis that two CK metrics, DIT and NOC, usually assume low values a statistical test is performed. This study also searches for the lowest threshold for both measures when the statistical significance of 0.05 is met. The importance of knowing the scale and the ranges of metrics used was discussed earlier in the study.

Sign test is a relatively simple statistical test, from which this research expects to generalize the results on the range of some CK metrics. Common statistical tests used in different research studies are outlined below.

The following tests are used to determine differences between independent groups:

- t-test for independent samples, for two samples when mean values for chosen variables are compared
- Wald-Wolfowitz test, the Mann-Whitney U test, and the Kolmogorov-Smirnov two-sample tests are nonparametric alternatives for the t-test
- ANOVA/MANOVA test is suitable for multiple groups
- Kruskal-Wallis analysis of ranks and the Median test are nonparametric alternatives for ANOVA/MANOVA test

To determine differences between dependent groups (the measures are taken from the same sample), the following tests can be used:

- t-test for dependent samples is suitable for comparing two variables measured in the same sample
- Sign test and Wilcoxon's matched pairs test are nonparametric alternatives to the t-test
- McNemar's Chi-square test is appropriate if the variables of interest are dichotomous in nature

- Repeated measures ANOVA is suitable when there are more than two variables measured in the same sample
- Friedman's two-way analysis of variance and Cochran Q test are nonparametric alternatives for more than two variables

This study uses the sign test for dependent groups, since CK measures come from the same samples. This test is comparatively simple in the computational sense, while capturing the information on the metrics range. A description of the applied sign test procedure is as follows.

For *n* pairs of data the sign test checks the hypothesis that the median of the differences in the pairs is zero. The test statistic is the number of positive differences. If the null hypothesis is true, then the numbers of positive and negative differences should be approximately the same. In fact, the number of positive differences will have a binomial distribution with parameters *n* and *p*.

This study uses sign test to investigate whether some software measures usually assume low values on an absolute scale. In this research, sign test is used for a single data set. For that purpose a threshold is introduced, against which values of two investigated measures are compared. If it is below the threshold a "–" is assigned to the project, otherwise a "+." Then the total number of "+" symbols is computed and the probability of getting such number or any lower number by chance is determined.

The probability *P(k)* of obtaining by chance exactly *k* projects with values lower than the chosen threshold is given by the binomial distribution:

$$P(k) = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k}$$

where:

- n is the total number of projects
- p is the probability of the value being lower than the threshold by chance (0.5)

The total probability *P* of obtaining by chance exactly *k* projects with values lower than the threshold is given by:

$$P = \sum_{i=0}^{i=k} P(i)$$

As with other methods in this study, the significance level of 0.05 is adopted, that is, if the total probability is above 0.05 the null hypothesis cannot be rejected.

The detailed results based on applying this test to the software data in this study are presented and discussed in the results section.

4.4 Decision trees

Decision tree learning is a widely used method for inductive reference. It is a method that approximates discrete-valued functions, and is especially suitable for modeling noisy data due to its robustness (Quinlan, 1993).

Although decision trees, just like statistical models, try to capture a function that maps each element of its domain to an element of its range, this approach is different from classical statistical methods. In the process of devising statistical models to represent the phenomena of interest, mathematical representations are obtained as results. In case of decision trees the idea is to obtain *rules* that lead to certain outcomes. Therefore, decision trees are a more flexible approach in categorizing possible output values without many assumptions of classical statistical approach.

Since the goal of decision trees is to predict or explain responses on a categorical dependent variable, this technique has much in common with the techniques used in the more traditional methods of discriminant analysis, cluster analysis, nonparametric statistics, and nonlinear estimation.

Creating decision trees based on software data and extracting rules from them is also investigated in this research. In that sense, two types of analyses are performed.

First, this study is concerned with finding rules that describe fault-proneness of classes in object-oriented software systems. In this case, classifiers are object-oriented software metrics from the CK metrics suite. For this purpose, the number of revisions, known for each class, is dichotomized, i.e. transformed into a binary number. Decision trees for different combinations of attributes and different sizes are grown, and the most successful rules are extracted.

Second, a NASA avionics data set on software problem reports is analyzed to create rules that would predict the time needed to fix the problems. The measures used are related to the number of modules to be changed, phase in the software development process where the problem occurred and the complexity of the problem.

Third, the investigation on the collinearity is undertaken, to determine whether this classification procedure “recognizes” it and discards all but one variable or whether this should be taken care of before including variables in tree generation.

Since the data under investigation shows high presence of zeros for output values, and classical statistical approaches do not deal with this problem very well, it is assumed that categorizing the data in only two groups would be the least strict. If that approach proves to be successful, more detailed categorization of output values can be tested.

The flexibility of decision trees makes them a very attractive analysis option, but this is not to say that their use is recommended to the exclusion of more traditional methods. Indeed, when the typically more stringent theoretical and distributional assumptions of more traditional methods are met, the traditional methods may be preferable. But as an exploratory technique, or as a technique of last resort when traditional methods fail, decision trees are, in the opinion of many researchers, unsurpassed.

Only few applications of decision trees in software engineering can be found. This fact is almost surprising considering a high number of studies trying to model software data. In addition, knowing that software data are usually considered statistically challenging; it comes as even more of a surprise that nonparametric methods like decision trees are not used more extensively.

A comprehensive study by Selby and Porter (1988) considers decision trees as a learning method based on which useful conclusions can be made in software projects. The analysis is based on 74 software metrics as attributes, including the separation between early development metrics and the rest of the available metrics for over 4700 objects. All the analyses are therefore done for the case of early known attributes and all attributes, and the results are compared.

This study attempts to achieve several important goals:

- Determine the most discriminative out of 74 available software attributes
- Investigate whether the early life-cycle measures are descriptive enough to base analyses only on them
- Predict total development effort and total number of faults in a software system

An important issue raised and investigated in this study is the complexity of the generated trees. The authors define the tree complexity as the number of unique attributes appearing in the top five levels of the tree. These attributes should provide the best differentiation among the objects, therefore representing the attributes the most useful to obtain. This issue is important with respect to the problem of overfitting data in case of exceedingly complex trees.

In their study, Porter and Selby (Porter and Selby, 1990) propose identifying components with high-risk properties, such as error-proneness and high development costs, using metric-based classification trees. The study is based on a large data set from a NASA project. Out of around seventy available software metrics, three were chosen to predict error-proneness and development costs of the components. Three metrics chosen are number of revisions for the component, its cyclomatic complexity and the type of the system the component belongs to. Since decision trees are suitable for categorical variables with low number of possible values, the data in this study was pre-processed to satisfy this condition.

The authors do not use any widely used software tool for automatic tree generation; instead they propose their own tool for that purpose. One of the characteristics of this tool is that classifiers are chosen based on the historic data from the company. This allows users of this tool to calibrate the parameters according to the specific nature of the project under investigation. The final results from the generated trees are rules that explain the nature of the phenomena in the software product in the study.

Briand and Wüst (1999c) attempt to combine Poisson regression and regression trees. The study investigates the problem of software cost estimation by combining these two approaches. Object-oriented design measures are chosen as classifiers in this study. The analysis of data is performed in two basic steps:

1. Regression trees are constructed based on the data
2. Stepwise Poisson regression is performed to improve the accuracy of the model

This approach is adopted in order to take advantage of modeling power of Poisson regression while still using the specific structures decision trees can capture. Finally, the accuracy of the derived model is assessed by applying cross validation procedure. The study concludes that the combination of Poisson regression and regression trees has helped to improve the prediction significantly. This can be explained by the fact that regression trees tend to capture the complementary structures in the data compared to Poisson model.

The concepts of decision tree learning are provided in the following sections.

4.4.1 Decision tree representation

Basic elements of a decision tree are root, internal nodes and leaf nodes. Root is the first node that splits the data set into categories. Internal nodes are places where further splitting occurs, according to the split criterion. Both root and internal nodes contain criteria according to which the data will be divided into specified categories.

Decision trees classify instances by sorting them from the root to the leaf or terminal nodes. Each node within a tree specifies a test of an attribute; therefore each branch descending from that node corresponds to one of the possible values for that attribute.

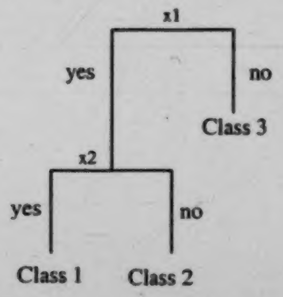

Figure 6: Decision tree - example

An example of a decision tree is shown in Figure 6. The tree contains two attributes, x_1 and x_2. At each tree node there is a split condition, according to which the output data is classified into different categories. In the presented example, for two attributes there are in total two split decisions to be made.

Attribute x_1 is the first that appears in the split criterion, indicating a good classifier. If the criterion is satisfied at this point further splitting is required depending on the value of the other attribute, x_2. If this second criterion is satisfied the terminal node labeled as "Class 1" is reached, which contains part of the data where both criteria were satisfied. When a terminal node is reached a rule can be extracted.

Following the same pattern, the following "rules" can be identified from Figure 6:

- If condition for x_1 is true and condition for x_2 is true then expected output is from Class 1
- If condition for x_1 is true and condition for x_2 is false then expected output is from Class 2
- If condition for x_1 is false then expected output is from Class 3

This example refers to the situation where the output variable assumes categorical values. In particular, decision trees are appropriate for data where the output variable has a small number of categorical values.

However, in reality and depending on the nature of the data, many more categories may be encountered for output variable. In addition, it is also common that the data is unbalanced, meaning that the percentile of output data might be unevenly distributed across categories.

In such situations, decision trees may not perform very successfully. One of the approaches to alleviate this problem is to restructure the output values so that the data is more balanced by merging classes at the output. The marginal case is, clearly, to create only two categories at the output. In the presented example, that would mean that instead of three, only two classes would remain.

In case of only two classes of output variable, it is easy to represent the problem in a geometrical way. An equivalent way of looking at this tree is that it divides the unit square as shown in Figure 7. When fixed-dimensional data have only ordered variables, the tree-structured procedure is a recursive partitioning of data space into rectangles. Assuming that the split conditions are $x_1 \leq 0.7$ and $x_2 \leq 0.5$ the graphical representations is shown in Figure 7.

From this geometric viewpoint, the tree procedure recursively partitions solution space into parts such that the populations within each part become more and more class homogeneous.

Suppose that the cases or objects fall into R classes. Assume that the classes are numbered with 1,2, ... , R and that C is the set of classes; that is

$$C = \{1,2, \dots , R\}$$

A systematic way of predicting class membership is a rule that assigns a class membership in C to every measurement vector $\mathbf{x}$ in X. That is, given any $\mathbf{x} \in X$, the rule assigns one of the classes $\{1,2, \dots , J\}$ to $\mathbf{x}$.

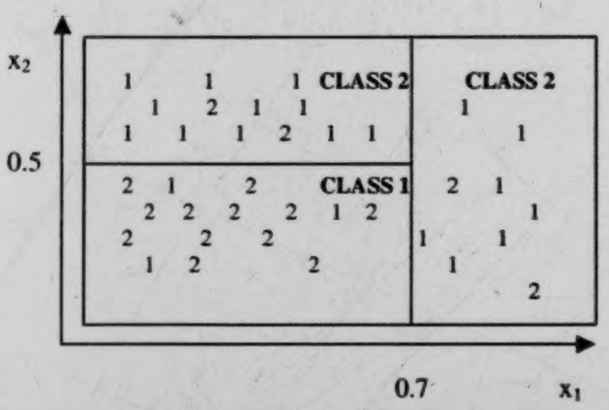

Figure 7: Graphical representation of a decision tree

In order to draw correct and accurate conclusions from decision trees, it is essential to choose good classifiers. Classifiers are usually constructed on the basis of past experiences. In systematic classifier construction, past experience is summarized by a learning sample. It consists of the measurement data on many cases observed in the past together with their actual classification.

In the context of software engineering this means that past data, including both internal and external software characteristics have to be known, in order to choose classifiers.

The essential issue in any type of tree growing procedure is how to identify good classifiers.

4.4.2 Designing the trees

In a classification model, the connection between classes and properties can be defined by something as simple as a flowchart or as complex and unstructured as a procedural manual. There are two very different ways in which they can be constructed. On one hand, the model might be obtained by interviewing the relevant experts; most knowledge-based systems have been built this way, despite the well-known difficulties attendant on this approach. Alternatively, numerous recorded classifications might be examined and a model constructed inductively, by generalizing from specific examples.

Whatever the procedure, the central idea in algorithms for finding best classifiers is to select the most useful attribute in each iteration of the algorithm. The most commonly used criterion for this selection is *information gain* measure, based on the entropy (Mitchell, 1997). In order to understand information gain as a selection criterion, the entropy is explained in more details.

Entropy characterizes the impurity of an arbitrary collection of examples. Given a selection *S* that contains different examples of a target concept, the entropy can be expressed as:

$$\text{Entropy (S)} \equiv \sum_{i=1}^{c} -p_i \log_2 p_i$$

where p_i is the proportion of *S* belonging to class *i* and *c* is number of categories at the output. Entropy assumes values in the range [0,1] where lower values are favored since they represent clearer subsets obtained from the original set *S*.

Given entropy as a measure of the impurity in a collection of training examples, information gain is defined as the expected reduction in entropy caused by partitioning the examples according to the chosen attribute.

Information gain, *Gain (S,A)* of an attribute *A* relative to a collection of examples *S* is expressed as

$$\mathrm{Gain}(S,A) \equiv \mathrm{Entropy}(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} \mathrm{Entropy}(S_v)$$

where *Values(A)* is the set of all possible values for attribute *A*, and S_v is the subset of *S* for which the attribute *A* has value *v* (i.e., $S_v = \{s \in S | A(s) = v\}$).

Information gain is the measure most commonly used by software tools for growing decisions trees to select the best attributes at each step in growing the tree. Clearly, attributes with higher values of information gain are selected first.

4.4.3 Optimal-sized tree

While growing a decision tree based on the underlying data, it is quite possible that a tree will overfit the data. The tree may have more structure than is helpful because it is attempting to produce several purer blocks where one less pure block would result in higher accuracy on unlabeled instances.

Overfitting occurs when it is possible to find an alternative hypothesis that fits the training example better than the original hypothesis. Hence, overfitting is a significant practical difficulty for decision tree learning, and the goal is to find the tree of the smallest size that would still produce successful rules.

The selection of overly large trees has led to much of the past criticism of tree structure procedures. To alleviate these problems, there have been several approaches.

First, the work was centered on finding appropriate stopping rules, that is, on finding a criterion for declaring a node terminal. However, after many attempts of devising acceptable stopping rules, finally the conclusion was reached that looking for the right stopping rule was the wrong way of looking at the problem. A more satisfactory procedure suggests growing a tree of unrestricted size and then pruning it upwards.

Pruning can be described as the process of selecting the "best" subtree in its size range from the originally grown tree. It consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node. Nodes are removed only if the resulting pruned tree performs no worse than the original over the validation set. Nodes are pruned iteratively, always choosing the node whose removal most increases the decision tree accuracy.

If this approach is followed, typically, as a tree is pruned upward, the estimated misclassification rate first decreases slowly, reaches a gradual minimum, and then increases rapidly as the number of terminal nodes becomes small.

4.4.4 Automatic generation of decision trees using software tools

Automated tree generation is certainly a shortcut in the effort-consuming process of selecting classifiers and the cutoff values.

Two well-known programs for constructing decision trees are C4.5 (Quinlan, 1993) and CART (Classification and Regression Tree) (Breiman *et al.*, 1984). Some other, general statistical tools, such as S-PLUS (http://www.splus.mathsoft.com/), also provide the facility for automatic tree generation. This study uses both S-PLUS and CART tools and compares results obtained from them. Important characteristics of both are provided.

Through different experiments, the following characteristics of S-PLUS have been noted.

First, the splits are univariate, which means that only one cutoff value at a time is a measure. In the literature there are examples when a combination of classifiers and their cutoff values can be a split condition for the data.

Second, at the split points there are only two outputs, in this case it means indicating whether the condition is satisfied or not. As a result, measures appear more times in a tree, since the tool attempts to refine the split conditions.

Third, two types of pruning are provided by S-PLUS: cost-complexity pruning and pruning to a specified tree size. For any of them, pruning can be done based on two criteria: minimizing the deviance as a measure of node purity, and minimizing misclassification rate.

However, although S-PLUS should produce optimal-sized trees based on the specified parameters, it is not always the case. This study uses pruning to a specified size while minimizing the deviance measure. When S-PLUS cannot produce tree of the desired size based on a minimized deviance, it returns the first larger tree that it finds to be optimal. Unfortunately, trees obtained this way are not always optimal, meaning that some splits are not really splits, because the outcome is the same no matter the split value. An example of a tree produced by S-PLUS when pruning to a desired size is shown in Figure 8 while Figure 9 presents manually adjusted optimal-sized tree.

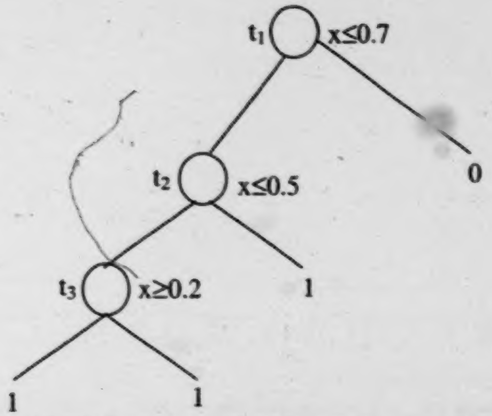

Figure 8: Example of automatically pruned tree

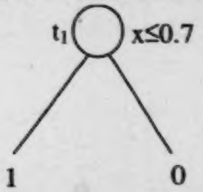

Figure 9: Optimized tree

From Figure 8 it is clear that the node t_3 does not actually classify data into different categories. This extends further to the node t_2, which means that as a real result there is only one node instead of previous three, and that the actual size of this tree is not four but two (Figure 9). Because of this and the fact that S-PLUS does not allow manual change in the tree, some additional work is required on the trees.

In order to assess experimental results in a well-controlled way, the following measures of tree size are introduced:

1. Desired size
2. Returned size
3. Actual size

Desired size is the size specified in the process of pruning trees. Typically, in the experiments all sizes from 2 up to the unrestricted size are chosen to experiment with.

Returned size is often equal and sometimes greater than the desired size (as explained in the section on automatic generation of decision trees). Actual size is the real size of the tree when non-optimal nodes are discarded.

For this analysis, the following experimental procedure is taken:

1. Select randomly a subset of 60% data points
2. Grow the tree of unrestricted size is grown
3. Prune the tree for all sizes starting with 2 up to the unrestricted size
4. Optimize the trees through manual adjustment
5. Validate results by testing the correctness of prediction using the remaining 40% of data for all tree sizes
6. Extract rules based on the nodes with the highest success rate

A further note here is that in some of the conditions, split values for the used metrics are not integers. In reality it is not possible since all chosen metrics assume values on an absolute scale. Therefore, these values are also adjusted on the process of extracting rules from the decision trees.

For example, if the rule states "*value* ≥ 0.5", the rule is adjusted to "*value* ≥ 1". Clearly, the condition "*value* ≥ 0" would not be correct in this case because "*value* = 0" does not satisfy the original condition. In case of the opposite sign (≤) the rule is just the opposite. In case of the condition "*value* ≤ 2.5" the adjusted condition is "*value* ≤ 3", since the original condition is still valid.

This study also investigates whether the proven presence of collinearity between some software measures influence the structure of the resulting tree. Since a high linear dependence between measures can jeopardize correctness of some statistical models, it is advised to exclude all but one of them from the prediction models.

This research investigates whether decision trees can recognize the presence of collinearity in data.

In order to investigate this, the experiments are performed on both industrial and artificial data sets with different data distributions, in order to determine whether a general conclusion can be drawn on this issue.

In conclusion, S-PLUS provides an automatic way of generating decision trees; however considerable manual adjustments are necessary in order to optimize them and extract rules from the trees.

Another tool used to build decision trees with software data is CART. Compared to S-PLUS, CART is more flexible offering a wide range of parameters to influence the generated outputs. The most significant advantage of CART over S-PLUS is that, when generating decision trees, it also takes the test sample into account. This alleviates the problem of overfitting, since the extracted rules need to be confirmed by testing samples. The most important parameters that can be chosen in CART are described here.

The rules for obtaining the "best tree" can be:

- Minimum cost tree regardless of the size
- Error within one standard deviation of the minimum cost tree
- Arbitrary chosen error rule

Minimum cost tree is similar to cost-complexity present in S-PLUS. It returns the most accurate tree representation. However, this accuracy usually comes at the expense of the size, meaning that such trees are typically too large. In order to partly reduce the size of the tree, the criterion of error within one standard deviation of the previous cost can be chosen instead.

The next parameter that can be specified arbitrarily is the method used to grow the trees. In case of classification trees (i.e. trees where attributes assume categorical values) the method can be:

- Gini
- Symmetric Gini
- Class Probability

Class Probability method is especially interesting, since software data can be quite unbalanced. It is difficult to say when in general one of these options is better. For some data sets, the difference is negligible, while in some cases it is up to 10% of the cost. The data investigated in this research show no difference when different method is used.

CART also offers different ways of assigning "importance" to classes of dependent variable. In that sense, the following options are possible:

- All categories have equal probability
- Probabilities match learning sample frequencies
- Probabilities match test sample frequencies
- The average of the probabilities coming from learning and testing sample is taken
- Probabilities match total sample frequencies

This feature introduces a significant improvement, since taking into account different probabilities of categories can again improve the process of obtaining optimal trees.

Finally, perhaps the most important advantage of CART over S-PLUS is automatic tree testing. First, the manual process of testing, which was required in case of S-PLUS, is no longer necessary. Second, the testing sample is also used in the process of producing trees, which automatically leads to smaller trees and better performance.

Again, the user faces several choices in testing procedure:

- No independent testing
- V-fold cross-validation
- Fraction of cases is randomly selected for testing purposes
- Test sample is contained in a separate file

When the first option of no independent testing is chosen, an exploratory tree is grown. It actually means that a maximal tree is produced since no validation verifies its performance. When this choice is made, CART generates very similar trees to those produces by S-PLUS.

V-fold cross-validation is convenient for small data sets since the process of randomly choosing training and testing subsamples is repeated arbitrary number of time.

The following testing procedure just randomly selects two thirds of data for training and the remaining third for testing.

Finally, there is the option to have training and testing samples in separate files, and to use the defined parts of data in this way. This way of testing is used in this research, since that enables proper comparison of the results obtained from both tools.

Although the two tools show some different characteristics, one feature is not provided in either of them. Both tools have univariate splits, meaning that the possibility of multiple outcomes from internal nodes does not exist. This particularly causes problems for S-PLUS since it requires more iterations of splitting which leads to more complex final rules. In case of CART this problem is not that essential because of the testing sample used while generating the trees.

Decision trees also have their drawbacks, especially regarding the ability to predict accurately, and the cost-effectiveness of its solutions. The process of testing the produced tree is particularly time consuming, since it is virtually impossible to create automatic procedures to run the data through the tree and thus calculate the performance of the tree.

However, their advantages including ease of use and interpretation, and their flexibility show the potential for a powerful tool for a wide variety of applications.

One more way to approach the issue of software quality and alleviate the problems stemming from software defects is software inspection.

4.5 Software Inspection

Since its conception about twenty years ago, software inspection has been considered a simple but nevertheless useful method of reviewing software documents. Software inspection techniques can, more or less, be applied in different stages of software development, starting as early as in the requirements phase and ending with the code inspection.

Many experimental studies have reported different aspects of software inspection and evaluated their applicability and usefulness in achieving higher quality of software products. This study suggests applying meta-analysis of effect sizes to assess whether the Ad-hoc reading technique is indeed less effective in defect detection than Perspective-Based Reading. Although there are many aspects of the software inspection process that can be investigated, the research is limited by the available data (Section 2).

The original inspection process was defined by Fagan (1976). Since then different variations of the procedure have been suggested; however, the idea is still the same. The most important techniques are overviewed briefly.

4.5.1 Software inspection techniques

The Fagan inspection procedure consists of the following phases: overview, preparation, inspection, rework and follow-up. The inspection begins with an *overview*, involving the entire team. After the introduction about the document, each team member carries out individual *preparation*, consisting of studying the document. The next stage is *inspection meeting* with all participants when the defects are being discovered. A *follow-up* then takes place, when all the required alterations are made, and the decision is made whether some parts of the documents need to be reinspected. Figure 10 outlines the phases of original Fagan inspection process.

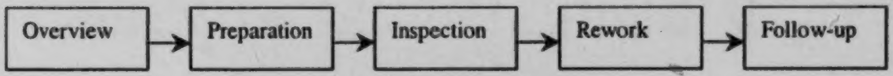

Figure 10: Fagan inspection process

The team size proposed for this type of inspection consists of four to six people with the following roles:

- Moderator, the person in charge of organizing the inspection process
- Author of the document being inspected
- Reader, who paraphrases the documents
- Recorder, who is in charge of noting the discovered defects
- Inspectors are the remaining persons participating in the process

Structures walkthrough is another widely used method of inspecting software documents proposed by Yourdon (1989).

The first phase of the procedure is *organization*, which begins with the request for a *walkthrough*. During that phase participants are getting prepared for the walkthrough by reviewing the product. The walkthrough itself begins with the presentation of the product and it should last between thirty and sixty minutes, and finish with a vote on the status of the product.

After the walkthrough phase is completed, a management summary and a list of detailed comments are prepared. In the *rework* phase the appropriate alterations are being made in the document, and finally a *follow-up* phase occurs to ensure that the required changes have been made to the product. The described inspection procedure is outlined in Figure 11.

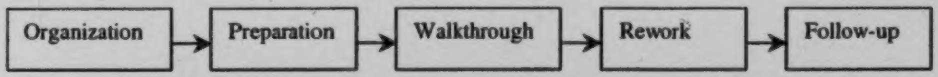

Figure 11: The Structured Walkthrough inspection process

There are seven types of participants in the structured walkthrough process:

- Coordinator, in charge of planning and organizing
- Presenter, who introduces the product to the participants
- Number of reviewers doing the defect detection

The inspection process described by Humphrey (1989) is in part similar to Fagan's inspection. The main difference occurs in the phase of preparation, where Fagan suggests that no defect detection technique should take place before the inspection phase.

Humphrey, on the contrary, introduces a new phase, called *analysis*, where the reviewers are asked to find and log defects. Then, these defect logs are passed to the analysis phase, and they are consolidated into a single list (Figure 12).

At the inspection meeting itself, each of the identified defects is discussed.

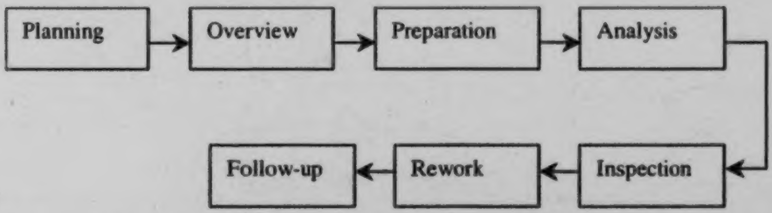

Figure 12: Humphrey's inspection process

One of the most comprehensive texts on software inspections is that of Gilb and Graham (1993). While this method is based on Fagan inspection and has similarities with Humphrey's inspection, it still has some specifics on its own (Figure 13).

The process begins with ensuring that some entry criteria are satisfied. This phase is followed by the *planning* phase, when the meeting is scheduled and the participants are chosen. *Kickoff* is the phase when the document is distributed and participants are assigned roles. The next phase, *checking*, is the time when each checker inspects the document himself. These potential defects are recorded for presentation in the next phase called *logging meeting*. Other potential defects can also be detected during the meeting, which can be followed by a *brainstorming* session. In the *edit* phase all the defects are compiled into a single list of defects and a *follow-up* is the next step. Finally, some *exit* criteria need to be satisfied in order to terminate the inspection process.

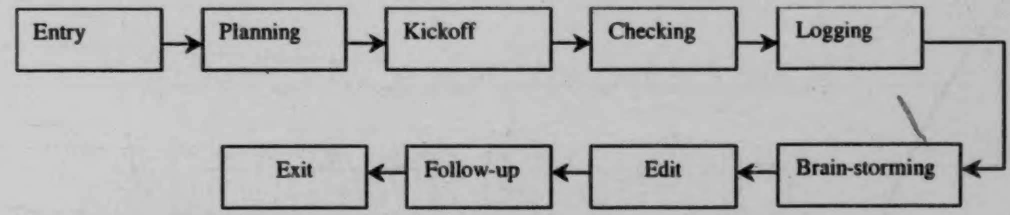
Entry
Planning
Kickoff
Checking
Logging
Brain-storming
Edit
Follow-up
Exit

Figure 13: The inspection process as described in Gilb and Graham

The three defined roles in this type of inspection are:

- Leader, in charge of organizing the whole process
- Author of the document, an invaluable source of explanations during the meeting
- Group of checkers who perform the review of the document

An alternative process of inspecting software documents, which does not require actual meetings with all participants, is asynchronous inspection (Figure 14). The idea of this approach is to avoid costly and organizationally complex meeting for document reviews. Instead, some means of supporting the discussion without requiring the presence of all team members are introduced.

Inspectors post "articles" that are later on read and edited by other participants. This procedure continues until a certain consensus about the quality of the document is reached.

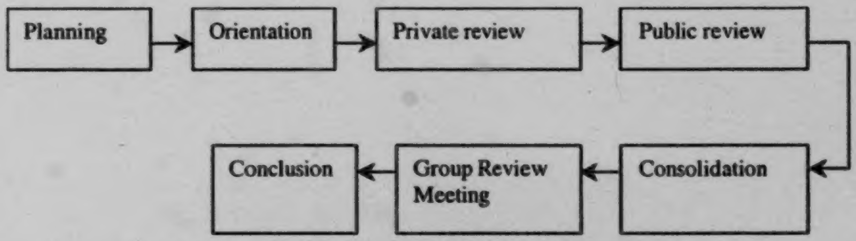
Planning
Orientation
Private review
Public review
Consolidation
Group Review Meeting
Conclusion

Figure 14: Asynchronous inspection process

An innovative approach in software inspection process was introduced with active design reviews (Figure 15). The main difference compared to the previous methods is that instead of one central meeting, more small meetings are held in order to evaluate separate parts and aspects of the inspected document.

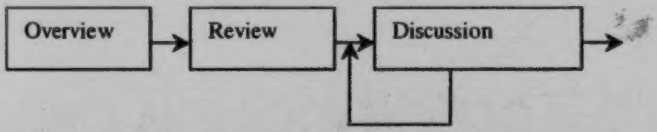

Figure 15: Active Design Review process

N-fold inspection represents the multiple inspection of the same document by different teams, in order to ensure that most of the defects are discovered. It is clear that by increasing the number of teams the percentage of defects should increase up to the point that all the defects are detected. The process is illustrated in Figure 16.

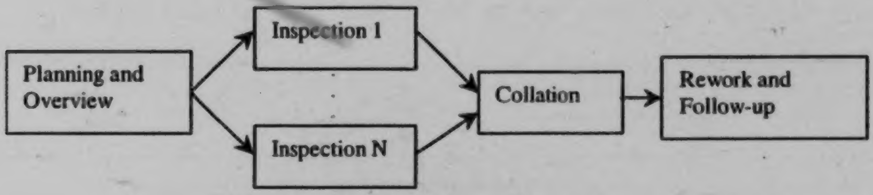

Figure 16: N-fold inspection process

However, a major issue related to this software inspection technique whether it is justifiable to use it, since there must be some overlap of the defects discovered by different teams. The issue of cost-effectiveness of different types of software inspection, and software inspection in general has been investigated in some experimental studies (Porter *et al.*, 1997).

4.5.2 Reading techniques for software inspection

Although each of the activities is important for a successful inspection, the key part of each inspection is the defect detection activity. Throughout this activity inspectors read software documents and check whether they satisfy quality requirements, such as correctness, consistency, testability or maintainability. Each deviation is considered a defect.

Because of its importance, adequate support for inspectors during defect detection can potentially result in dramatic improvements in inspection effectiveness and efficiency.

Different methods have been proposed to help the reviewers accomplish the task of finding the defects with less difficulty in the process of reading software documents.

Ad-hoc reading is the technique that does not provide any explicit advice for inspectors on how to proceed, or what specifically to look for during the reading activity. Hence, inspectors must resort to their own intuition and experience to determine how to go about finding defects in a software document.

Checklists offer stronger support mainly in the form of yes/no questions that inspectors have to answer while reading a software document. Although more helpful than ad-hoc reading, checklists suffer from several shortcomings among which the most important are:

- Checklists are often based on the past defect information, which potentially makes them limited for future studies
- They often contain too many questions without clear guidance how to answer them
- Inspectors are not required to document the analysis, which leads the results to be unrepeatable from other reviewers

A more recent approach, *scenario-based reading*, has been proposed to tackle some of the deficiencies of the previously used methods. The basic idea of scenario-based reading is the use of so-called scenarios that describe how to go about finding the required information in a software artifact as well as what that information should look like.

Most commonly used reading techniques are:

1. Defect-based reading (DBR) focused on defect detection in requirements, where the requirements are expressed using a state machine notation
2. Perspective-based reading (PBR) also focused on defect detection in requirements, but for requirements expressed in natural language
3. User-based reading (UBR) concerned with anomaly in user interfaces

A recent study by (Travassos *et al.*, 1999) proposes a new set of software reading techniques for early object-oriented design, named *Traceability-Based Reading (TBR)*. A major difference between the previously proposed reading techniques and TBR is that TBR proposes that checking for the correctness in the documents must be twofold. As in requirements inspection, the correctness and consistency of the design diagrams themselves must be verified to ensure a consistent document. This is referred to as "horizontal reading". On the other hand, it is also necessary to verify the consistency between design artifacts and the system requirements. This is achieved using so-called "vertical reading".

4.5.3 Experimental studies on software inspection

Software document inspection has been a generous resource for many experimental studies in software engineering field. Keeping in mind the many different proposed techniques, many research questions are introduced and discussed.

Typically, studies are aimed to determine efficiency and effectiveness of the proposed methods subject to different parameters and constraints. The list of possible parameters and attributes to consider is virtually endless. Some of the variables according to the type are presented in Table 15.

Type	**Variable**
Method	Method used
Material	Size of the material
	Type of the material
Time	Length of the meeting
	Defect discussion time
	Preparation time
	How late in software development the inspection was done
People	Number of participants
	Level of author skills
	Level of inspectors' skills
	Experience in the company
	Experience on the project

Table 15: Possible variables in empirical studies on software inspection

A study by Porter *et al.* (1997) draws several conclusions on software inspection practice in a long-term experiment in industry. First, it concludes that one person is much less effective in defect detection than two- and four-persons teams, and that two- and four-person teams perform almost equally. Second, they prove that multiple sessions do not contribute to discovering higher number of defects compared to a single session. Third, they confirm that repairing defects between multiple sessions has no effect on observed defect density. Finally, the study investigates whether a better organization in the planning phase can improve the effectiveness of the overall inspection process.

A software document reading technique using checklists is analyzed in (Chernak, 1996). The paper focuses on a statistical approach for formal synthesis and improvement of checklists. The main goal of the study is to determine the best way to create checklists for specific document in the inspection. To achieve this statistical and empirical approaches are taken. Finally, some guidelines how to improve the checklists in general are suggested.

Different reading techniques are also assessed in (Porter *et al.*, 1995). The study concludes that ad-hoc reading and using checklists are similarly effective, while scenario based techniques are more effective. The authors also evidence that collection meetings do not contribute significantly to the number of defects discovered.

Similar results are obtained in (Porter and Votta, 1994) on the basis of inspection of the software requirements specifications.

N-fold inspection procedure is investigated in (Schneider and Martin, 1992) to assess the defect detection in user requirements documents, since the flaws discovered in early stages of software development are cheaper to correct. The experiment described in this study is based on injected defects of specific type. Nine independent inspection teams were given user requirements documents with instructions to locate as many faults as possible. The analysis shows that a single team is able to detect no more that 35% of present faults, where the use of N-fold inspection improves this result to as high as 78%.

A novel family of reading techniques for software inspection was proposed by Travassos *et al.* (1999). The proposed techniques are suited for early object-oriented design. The study was conducted in the university setting. The first conclusion indicates that syntactic correctness is more important while inspecting object-oriented design documents, due to the number of separate but inter-related diagrams that must be kept consistent. The second important result is concerned with the proposed so called "vertical" and "horizontal" reading techniques. The results indicate that different reading technique leads to different types of defects to be discovered.

The importance of meetings is in detailed discussed in (Porter and Johnson, 1997). The study is based on two experiments undertaken in university setting. The results indicate two conclusions. First, with respect to defect detection effectiveness, typical meeting-based software review methods are neither substantially more effective nor less effective than nonmeeting-based software review alternatives. Second, individual defect detection typically generates far more issues than group-based defect detection, yet has the cost of higher false rates and higher issue duplication.

The importance of meetings is also addressed by Seaman and Basili (1997) focusing on the communication between the participants in the process. This study evidences that two major reasons for longer meetings are the time spent in defect detection activity and time spent discussing general issues. Second, it discovers that the more the inspection participants interact with each other on a regular basis, the fewer defects are reported. Third, the more complex the inspected material is, the fewer defects will be found and less time is invested on discussing general issues. Fourth, the more the inspection participants have worked together in the past or still working together, the less time they will spend discussing general issues. Finally, the later in the project the inspection occurs, the less time will be spent discussing general issues.

The study by Briand *et al.* (1997) is concerned with building efficiency benchmarks to enable the comparison between different inspection techniques. It is meant to guide companies create and adjust their own enterprise-wide benchmarks. Such benchmarks should help companies determine whether its inspection process has a satisfactory efficiency level or whether corrective actions are needed.

Variations in results coming from software inspection experiments are investigated in details in (Porter *et al.*, 1998). The study attempts to determine the influence on defect detection effectiveness and inspection interval resulting from changes in the structure of the software inspection process. The effects of team size, number of sessions and whether the defect repairs make any difference are accounted for. However, it is questionable whether the results can be generalized, since the data come from one company, and only one project written in one programming language.

4.6 Validation of results

Whatever the approach is chosen to model software data, it is important to assess or validate the obtained results. When a model is chosen because of qualities exhibited by a particular set of data, predictions of future observations that arise in similar fashion will almost certainly not be as good as might naively be expected.

The purpose of the validation process is to select the model that will give the best results on the future data, which means that it should not be too sensitive depending on the specifics of the used data set.

The idea of validating the models was originally conceived by statisticians who tried to select the best regression model for predictive purposes. Since then many methods have been proposed and discussed (Stone, 1974). For different purposes, different methods are most suitable, but one conclusion is common to all of them – any type of validation is still better than deriving a model without a chance to assess its accuracy and applicability in general.

The most common statistical criterion used to achieve this purpose is cross-validation. In its most simple but nevertheless useful form, it consists of the controlled or uncontrolled division of the data sample onto two subsamples. These samples are referred to as "construction" or "training" and "validation" or "testing". From that point they differ in order to accommodate the needs of different analyses.

The methods proposed in the literature are outlined here.

One of the versions of cross-validation, quite often used is *Leave-k-out* validation. The procedure for applying this type of assessment is quite straightforward, consisting of the following steps:

1. k data points are randomly chosen from the data sample and set aside
2. Investigated model is optimized based on this sample
3. Model is validated using the remaining portion

A common approach is to split data in parts consisting of 60% of data for constructing the model and the remaining 40% for testing it. Alternatively, 2/3 and 1/3 of the data can be used in the similar fashion.

Some studies discuss whether the choice of *k* data points to be left out should be really random. They argue that the chosen subsample does not necessarily represent the true nature of the data and that some improvements could be made on this issue.

An innovative approach of *active pattern selection* is suggested, with the idea of selecting the training set in the way to obtain more reliable results (Leisch *et al.*, 1998).

The special case of the previous method is to erase only one data point from the whole data set in the *Leave-one-out* method. This procedure is repeated so that each data point is omitted once. Therefore, the total of *n* models are devised, where *n* is the size of the data set. Again, the obtained models are averaged so as to obtain the final result.

Leave-one-out validation is more appropriate for smaller data sets, since splitting the data into two parts of comparable sizes can reduce the significance of the achieved results, thus jeopardizing their validity. Leave-one-out cross validation has a smaller bias than leave-k-out with $k>1$, since there is no problem with the selection of k data points that need to be eliminated for the training part. However, a major problem present in this approach is its computing intensive nature, since the whole process needs to be repeated *n* times. Depending on the size of the data set and the number of repeated experiments, it can be quite costly.

v-fold cross validation is one way to improve validation methods based on holding out a portion of data. The data set is divided into *v* subsets, and the leave-k-out method is repeated k times. Each time, one of the *v* subsets is used as the test set and the other *v-1* subsets are put together to form a training set. The average error across all *v* trials is then computed.

The advantage of this method is that the importance of how the data is split is less important. Every data point gets to be in a test set exactly once, and gets to be in a training set *v-1* times. Naturally, the variance of the resulting estimate is reduced as *v* is increased.

The disadvantage of this method is that the training algorithm has to be rerun from scratch *v* times, which means it takes *v* times as much computation to make an evaluation. Also, it is not suitable for small data sets.

A variant of this method suggests randomly dividing the data into a test and training set *v* different times. The advantage of doing this is that the size of test set and the number of trials to average results can be independently chosen. The detailed description of this method is outlined here:

1. Data is split randomly in two parts, learning set of size n_0 and a test set of size m_0, where $m_0 + n_0 = n$, and n is the size of the whole data set
2. This procedure is repeated arbitrary number of times
3. For each split, estimates are developed based on the data in the learning set
4. Estimates are then tested on the data in the test set
5. Finally, the results obtained from each experiment are averaged

v-fold cross-validation and repeated training and testing approach can be combined in the *Repeated v-fold cross-validation*. It is quite simple if the two incorporated methods are known; it consists of repeating *v*-fold cross-validation procedure many times, as in the repeated training and testing approach. Finally, the simple average is computed out of all the repeated iterations.

As mentioned in the introductory part of the validation section, different methods are suitable for the specific goals to be achieved, and the characteristics of the data sets investigated. The major advantage of cross-validation is that it is applicable to a wide variety of problems.

For different purposes, this study has leave-k-out, with 60% of data for training and remaining 40% for testing purposes. The detailed results as well as the discussion of the performance of these are discussed in the results section.

Apart from these types of validation, a different validation is applied in the meta-analytic methods applied in this research. The details of this approach are explained in the section describing these meta-analytic techniques.

5. Empirical evaluation of the proposed quantitative software engineering methods

In this section, the results are provided and discussed for all the experiments undertaken. Since the description of the data used in the study and the background explanation for all the methods applied are presented earlier in the study, this section outlines the experimental results and discussion where necessary.

5.1 Meta-analytic study of object-oriented systems

Meta-analytic technique of Weighted Estimators of Common Correlation is applied for two purposes in this research.

First, it investigates collinearity between some CK metrics as internal software attributes.

Second, it analyzes correlations between internal measures and the external software measure aimed at identifying best performing CK measures.

5.1.1 Collinearity in software data

The issue of collinearity is analyzed on both public domain data sets (100 Java and 100 C++ projects) and two consecutive releases of industrial data. The meta-analytic technique is based on correlation coefficients calculated between all pairs of CK metrics. Non-parametric Spearman correlation is used in the analyses, since it does not assume any specific data distribution. In general, non-parametric methods are less powerful; however, they are more flexible about the stringent conditions related to the data behavior.

Table 16 summarizes weighted correlations for public domain data sets. Considering that the value 0.5 is chosen as the indication of high correlation, the results confirm that the three CK metrics, RFC, CBO and NOM, are linearly dependent. In case of Java projects, the highest correlation appears between RFC and CBO, and second between RFC and NOM. In C++ projects the situation is just the opposite. In both Java and C++, the correlation between CBO and NOM is the lowest. Boldface letters indicate correlations of interest for this study, and those in italic show correlations that are also high but appear as a byproduct.

		NOM	DIT	NOC	CBO	RFC	LCOM
Java Projects	NOM	1	0.03	0.18	**0.57**	**0.82**	*0.99*
	DIT		1	-0.03	0.26	0.21	0.04
	NOC			1	0.03	0.08	0.19
	CBO				1	**0.87**	0.56
	RFC					1	*0.81*
	LCOM						1
C++ Projects	NOM	1	0.46	0.24	**0.59**	**0.97**	*0.92*
	DIT		1	0.12	0.38	0.50	0.47
	NOC			1	0.04	0.21	0.24
	CBO				1	**0.72**	0.49
	RFC					1	*0.85*
	LCOM						1

Table 16: Summary of Weighted Correlations for public domain data sets

In order to check whether these results can be considered valid, i.e. whether there is enough grounds to pool the common conclusion based on the underlying data, the test of homogeneity is performed. For this purpose chi square statistics is used as proposed in (Hedges and Olkin, 1985).

Q values are compared with $Q_{0.05}$, and if the calculated Q is lower than $Q_{0.05}$ the conclusion is that the data are homogeneous for that particular relationship. Table 16 contains the results of the homogeneity test.

		DIT	NOC	CBO	RFC	LCOM
Java Projects $Q_{0.05}$= 6291.61	NOM	3188.16	401.82	**1404.50**	**5236.62**	6553.93
	DIT		2095.26	1153.95	2726.33	2276.49
	NOC			254.41	398.97	395.09
	CBO				**2064.25**	1016.83
	RFC					2801.56
C++ Projects $Q_{0.05}$= 6171.21	NOM	1629.97	431.64	**1187.20**	**1417.07**	12932.93
	DIT		326.33	947.39	935.21	1168.64
	NOC			339.61	321.52	429.93
	CBO				**698.89**	683.62
	RFC					1366.98

Table 17: Summary of Homogeneity Test for public domain data sets

As evidenced in Table 17, the homogeneity test is satisfied for all three relationships under consideration. Considering that the data are not coming from the controlled experiments, but rather easily available public projects, the results of the homogeneity tests are very good, strongly confirming the set hypotheses.

The correlations from Table 16 also indicate very high correlation coefficients in pairs (NOM, LCOM) and (RFC, LCOM). In addition, the correlations between NOM and LCOM are higher in Java projects than in C++. The higher correlation in Java projects could be explained with the presence of accessor methods, more common in Java than in C++. A feature of Java is that for every "relevant" attribute in a class, there should be *get* and *set* methods for accessing such attribute. In case of *n* attributes in a class, there will be typically *2n* accessor methods. Clearly, this results in an increased number of methods for a class. As each couple {*get*, *set*} does not have any common variable with any other couple {*get*, *set*}, LCOM tends to increase dramatically with the number of {*get*, *set*} methods.

However, the relationships between NOM and LCOM cannot be confirmed, as evidenced in Table 17. The high values of the correlations between RFC and LCOM are considered a byproduct of the high correlations between NOM and LCOM.

The same analysis is performed on both releases of industrial data. Table 18 presents the summary of correlations and Table 19 the results of homogeneity test.

		NOM	DIT	NOC	CBO	RFC	LCOM
First release	NOM	1	0.35	0.38	**0.62**	**0.89**	*0.87*
	DIT		1	-0.01	0.41	0.39	0.30
	NOC			1	-0.05	0.02	0.06
	CBO				1	**0.81**	0.58
	RFC					1	*0.81*
	LCOM						1
Second release Without Common Software	NOM	1	0.24	-0.05	**0.53**	**0.93**	*0.95*
	DIT		1	-0.11	0.14	0.19	0.25
	NOC			1	-0.02	0.01	0.01
	CBO				1	**0.59**	0.40
	RFC					1	*0.89*
	LCOM						1
Second release With Common Software	NOM	1	0.27	-0.04	**0.54**	**0.94**	*0.95*
	DIT		1	-0.09	0.21	0.27	0.32
	NOC			1	-0.07	-0.02	0
	CBO				1	**0.83**	0.34
	RFC					1	*0.75*
	LCOM						1

Table 18: Summary of Weighted Correlations for industrial data sets

The results (Table 18) again confirm the existence of the investigated collinearity. Also, they indicate that when the Common Software part is introduced, the homogeneity becomes more critical issue. Nevertheless, the homogeneity is satisfied for the correlations of interest.

		DIT	NOC	CBO	RFC	LCOM
First release $Q_i = 9.49$	NOM	9.28	8.66	**9.15**	**9.24**	4.64
	DIT		4.87	4.27	10.98	71.65
	NOC			9.47	9.38	11.41
	CBO				**6.09**	9.01
	RFC					4.55
Second release Without Common Software $Q_i = 12.59$	NOM	12.57	12.35	**7.36**	**9.29**	62.05
	DIT		3.93	6.14	9.31	8.41
	NOC			7.84	8.41	12.23
	CBO				**6.78**	8.45
	RFC					9.28
Second release With Common Software $Q_i = 14.07$	NOM	13.95	12.57	**7.51**	**9.79**	64.98
	DIT		4.24	13.93	21.54	16.71
	NOC			11.95	10.12	13.07
	CBO				**13.27**	11.52
	RFC					23.98

Table 19: Summary of Homogeneity Test for industrial data sets

In conclusion, the results of the detailed analysis of the problem of collinearity between NOM, RFC and CBO is confirmed for both public domain and industrial software data, therefore presenting a significant result. In case of linear and log-linear models, only one of the collinear CK metrics should be used in building them.

5.1.2 Meta-analysis for identifying best predictor variables

For the industrial data sets the same method is used to identify the best predictors in modeling defect software behavior. This time correlation coefficients are calculated between each CK metric and the number of modifications for each class. The goal of such an analysis is to examine whether there are metrics that are, in general, more highly correlated to the number of defects (or alternatively number of modifications). Such metrics would present better predictors in analysis of software behavior.

Table 20 shows individual weighted correlations for each project in both releases, sizes in terms of number of classes (n) and LOC for each project.

		LOC	**NOM**	**DIT**	**NOC**	**CBO**	**RFC**	**LCOM**	**n**	**LOC/proj**
Release 1	Project 1	0.07	0.14	-0.07	-0.20	0.00	0.11	0.11	93	8802
	Project 2	0.24	0.23	0.59	-0.13	0.36	0.31	0.24	120	6496
	Project 3	0.46	0.41	0.24	0.03	0.43	0.43	0.46	101	25325
	Project 4	0.36	0.31	0.06	-0.21	0.45	0.45	0.32	38	17791
	Project 5	0.24	0.18	0.05	-0.17	0.19	0.25	0.16	44	5316
Releases 2	Project 1	0.10	0.16	0.10	-0.08	-0.09	0.17	0.16	247	19238
	Project 2	-0.03	0.10	0.15	0.35	0.17	0.05	-0.04	21	2212
	Project 3	0.46	0.23	0.13	0.16	0.21	0.31	0.35	71	9646
	Project 4	0.13	0.04	0.08	-0.02	0.16	0.06	0.01	215	8330
	Project 5	0.38	-0.08	0.15	0.22	-0.01	-0.06	-0.02	33	1388
	Project 6	0.37	0.37	-0.20	0.47	0.06	0.31	0.26	16	287
	Project 7	0.16	-0.18	-0.20	0.20	0.19	-0.19	-0.18	68	1064
	Common Software	0.12	-0.01	-0.21	0.07	-0.04	0.03	-0.02	575	30921

Table 20: Summary of individual weighted correlations for each project

Table 20 provides both correlation coefficients for the whole releases and both releases together as well as chi square statistics. Significance level of 0.05 is used to check homogeneity. The values marked in italic present the cases where homogeneity could not be confirmed. In addition, the chi square test for significance level of 0.01 is also provided to show that the homogeneity, if critical, can be compared to this, less strict, value.

			LOC	NOM	DIT	NOC	CBO	RFC	LCOM	Q test	
										α=0.05	α=0.01
First release		Correl.	0.29	0.27	0.27	-0.12	0.30	0.32	0.28		
		Homog.	9.16	4.62	*33.67*	3.33	*13.13*	7.08	8.10	9.49	13.28
Second Release	Without CSW	Correl.	0.15	0.10	0.09	0.00	0.05	0.12	0.10		
		Homog.	12.04	10.30	6.75	*13.28*	10.93	12.10	*13.75*	12.59	16.81
	With CSW	Correl.	0.08	0.04	-0.07	0.05	0.17	0.07	0.04		
		Homog.	12.93	12.97	*31.12*	13.76	*14.54*	14.00	17.10	14.07	18.48
First and Second release	Without CSW	Correl.	0.21	0.15	0.14	-0.02	0.15	0.18	0.16		
		Homog.	*24.18*	*22.73*	*50.33*	*22.05*	*37.13*	*29.94*	30.50	19.68	24.73
	With CSW	Correl.	0.18	0.10	0.01	0.01	0.08	0.13	0.10		
		Homog.	*27.53*	*32.19*	*96.15*	*25.93*	*51.14*	*39.06*	*41.76*	21.03	26.22

Table 21: Summary of correlations and homogeneity values for telecommunication data

The results indicate that for several CK metrics, the correlation against number of modifications as the external measure is quite similar (LOC, NOM, CBO, RFC). While such results do not facilitate drawing conclusions on best predictors, they are partly expected due to the collinearity between these metrics.

If the results are examined more closely, it can be noted that in three analyses (first release, second release without and with common software) RFC and CBO consistently show higher correlation than other metrics. This also does not come as a surprise, since both RFC and CBO are shown to be good predictors in similar studies (Succi *et al.*, submitted to IEEE Transactions of Software Engineering).

The results also evidence that introducing common software in analysis as an additional data set of the second release, lowers the pooled correlations and worsens the homogeneity values. Since common software is the largest "application", this effect cannot be ignored.

For the first release, the results indicate that the homogeneity is satisfied for LOC, NOM, NOC, RFC and LCOM at the 95% level, and for CBO at 99% level. For DIT, the homogeneity is not satisfied.

For the second release, the results show that the homogeneity is satisfied for LOC, NOM, NOC and RFC at the 95% level, and for CBO and LCOM at 99% level. Again, for DIT, the homogeneity is not satisfied.

When two releases are combined, the homogeneity is not confirmed for any of the measures. Consequently, such meta-analysis cannot be valid.

In conclusion, this method shows that there is no single CK metric that significantly outperforms any other for predicting software faults. Several of the measures show good potential; however, since they are linearly dependent this is not an unexpected result.

5.2 Analyzing measures assuming low values on an absolute scale

Knowing how to treat measures on different scales is an important issue raised and discussed in the section on software metrics.

All CK metrics are on an absolute scale, and two of them, DIT and NOC, are proved to assume values in narrow range (Tang *et al.*, 1998; Ronchetti and Succi, 1999). This research is aimed to prove or disprove this hypothesis using public data sets described previously. The hypotheses to test are as follows:

- H_1: The maximum values of NOC and DIT are not below 10 in the analyzed C++ Projects
- H_2: The maximum values of NOC and DIT are not below 10 in the analyzed Java Projects.

It is the goal of this experiment to accept or reject this hypothesis at the 0.05 level.

In order to gain deeper understanding of data ranges, the first step is to investigate the distributions of both measures in both Java and C++ projects. The boxplots in **Error! Reference source not found.** represent the distributions of both measures in both Java and C++ projects.

Boxplots are a way of summarizing a distribution of several data sets. The "box" in a boxplot shows the median score as a line and the first (25th percentile) and third quartile (75th percentile) of the score distribution as the lower and upper parts of the box. Thus, the area in the "box" represents the middle 50% of the data. The "whiskers" above and below the boxes represent the largest and smallest observed scores that are less than 1.5 box lengths from the end of the box. In practice, these scores are about the lowest and highest values one is likely to observe. Finally, there are outliers, representing the values that are outside the mentioned ranges. By portraying the scores for more than one group next to each other, one can frequently see the major trends in the dataset

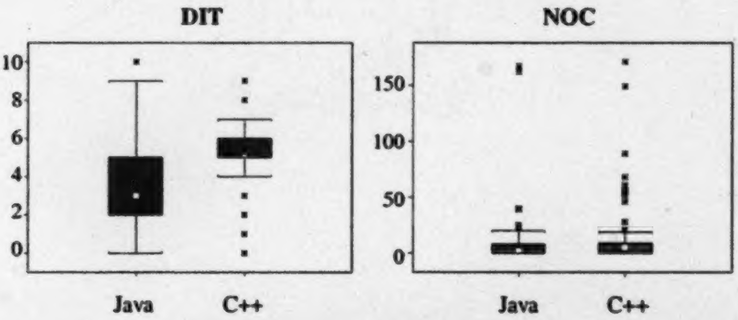

Figure 17: Distributions of the maximum values of DIT and NOC in public domain data sets

From the distributions of the data, several conclusions can be drawn:

- Maximum values vary more for DIT in both Java and C++ projects
- The range of maximum values for NOC is quite narrow and similar for both programming languages; however, there is a higher number of outliers.
- Maximum values are slightly higher for C++ than for Java projects, and the presence of outliers is more significant in C++ projects.

Statistical sign test is used in order to confirm or disconfirm the chosen hypotheses. Table 22 shows total probabilities of obtaining maximum values below 10 by chance.

	DIT	NOC
Java	0.00	1.35E-10
C++	3.98E-27	9.05E-08

Table 22: Probabilities of obtaining values for DIT and NOC lower than 10 by chance in public domain data sets

Since the chosen statistical significance for this analysis is also 0.05, the results indicate that this probability is far below this threshold, meaning that it is certain that DIT and NOC in general assume values lower than 10.

Although the threshold 10 is satisfied in case of both programming languages and both CK metrics, the analysis is broadened in order to identify the lowest thresholds for each of the combinations metric - project type.

In order to achieve this goal, the same procedure of testing hypotheses using sign test is repeated, but instead of threshold equal to 10, several values of thresholds are considered (say 10, 9, 8, etc.) while still taking into account the adopted significance level of 0.05. Table 23 outlines the corresponding probabilities.

The detailed analysis shows that for different programming languages and DIT and NOC, different thresholds can be set. The lowest threshold achieved is 4 in case NOC–Java and the highest is 6 in case of DIT–Java and NOC–C++.

Threshold	DIT – Java	DIT – C++	NOC – Java	NOC – C++
9	7.97E-29	3.22E-24	9.05E-08	2.35E-06
8	1.31-25	6.26E-23	8.34E-07	1.76E-03
7	9.56E-16	1.27E-16	6.29E-06	6.66E-02
6	6.29E-06	1.044E-10	9.16E-05	1.35E-10
5	1	2.04E-04	8.95E-04	0.90
4	1	0.24	3.32E-03	0.99
3	1	1	0.31	1

Table 23: Results of the Sign Test for different values of the threshold

The analysis confirms the hypothesis that both measures (DIT and NOC) assume values in narrow ranges, in general. This conclusion is important since measures with such characteristics, when used as variables in statistical modeling, should be treated with care. However, the reached conclusion does not claim that this is a good or bad practice, since these measures are not related to any external software representation of its behavior. It only confirms the state of the practice in following object-oriented concepts.

5.3 Combining regression models across software projects

The following section presents the results of two methods that investigate the feasibility of combining regression models with software data. The model is applied only to industrial data, since for public domain data no external measure was available.

The analyses are based on univariate models with RFC, since RFC is an early-cycle measure, proven to be a good predictor in some experimental studies (Succi *et al.*, submitted to Transactions of Software Engineering). Multivariate models would be more complex to investigate due to the increased dimensionality of the models and possible problems caused by the collinearity would have to be considered.

Combining regression models where errors are correlated is possible using Seemingly Unrelated Regression Equations method (SURE), as discussed in the background section of this method. Since the prerequisite for applying this method is existence of the correlation between errors of regression across projects, the actual correlations are calculated first.

	Project 1	Project 2	Project 3	Projects 4
Project 1	1	0.10	0.05	0.04
Project 2		1	-0.03	0.35
Project 3			1	-0.20
Project 4				1

Table 24: Summary of correlations between errors for the first release of telecommunication data

	Proj 1	Proj 2	Proj 3	Proj 4	Proj 5	Proj 6	Proj 7	CSW
Project 1	1	-0.15	-0.02	-0.32	-0.20	0.03	-0.35	0.11
Project 2		1	0.28	0.46	-0.30	-0.20	-0.36	0.05
Project 3			1	-0.05	0.20	-0.15	0.12	0.15
Project 4				1	0.07	0.14	-0.38	0.31
Project 5					1	-0.01	0.41	0.00
Project 6						1	-0.01	0.11
Project 7							1	-0.53
CSW								1

Table 25: Summary of correlations between errors for the second release

The results from both releases evidence that some correlation exists between the errors.

Moderate correlation exists between projects 2 and 4, and 3 and 4 of the first release, and projects 1 and 4, 1 and 7, 2 and 4, 2 and 5, 2 and 7, 4 and 7, 5 and 7, 5 and Common Software, and 7 and Common Software of the second release. Negative correlations can also be "high" since they also prove the existence of relationship between variables. Although the correlations are not very high, the analysis shows that the dependence of errors cannot be denied. Therefore, the application of this method is justifiable.

The presence of correlated errors is addressed with the covariance matrix of errors, since the covariance describes variables that change in the same fashion. That is the reason why the real benefits of this model may not be so evident on the data set used in this study.

As explained in the description of SURE models, time dimension of the data is necessary to provide their proper ordering. The measure used for this purpose is the date of the latest revision for each file. Considering that only files containing one class are included, this date is also the latest date of revision for the corresponding class. Also, due to the different sizes of data sets, the extension to the original method is used to account for this.

Project 1	Number of Fixes = 0.012*RFC + 2.431
Project 2	Number of Fixes = 0.008*RFC + 0.169
Project 3	Number of Fixes = 0.012*RFC + 1.143
Project 4	Number of Fixes = 0.008*RFC + 1.354
Release 1	**Number of Fixes = 0.009*RFC + 1.696**

Table 26: Regression models from the first release of telecommunication projects

Results in Table 26 and Table 27 show individual linear regression models with RFC and the pooled models using SURE approach.

Figure 18 illustrates univariate regression models for the data in the first release. The resulting regression line is indicated in the figure.

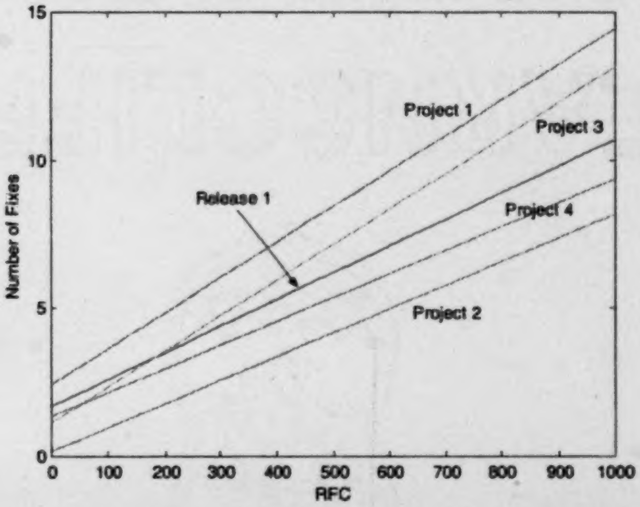

Figure 18: Illustrated regression models for the first release

The results from the second release are presented in Table 27. As discussed previously, Common Software is treated as a separate application, and its inclusion in analyses should be done with caution. For that reason, the second release is analyzed without and with Common Software included.

Project 1	Number of Fixes = 0.011*RFC + 1.036
Project 2	Number of Fixes = 0.010*RFC + 0.274
Project 3	Number of Fixes = 0.024*RFC + 0.362
Project 4	Number of Fixes = 0.054*RFC + 1.711
Project 5	Number of Fixes = -0.007*RFC + 0.263
Project 6	Number of Fixes = 0.016*RFC + 1.283
Project 7	Number of Fixes = 0.009*RFC – 0.036
Release 2 without CSW	**Number of Fixes = 0.010*RFC + 0.105**
CommonSW	Number of Fixes = 0.017*RFC – 0.096
Release 2 with CSW	**Number of Fixes = 0.075*RFC – 0.890**

Table 27: Regression models from the second release of telecommunication projects

Figure 19 represents individual and combined regression lines based on the data in the second release.

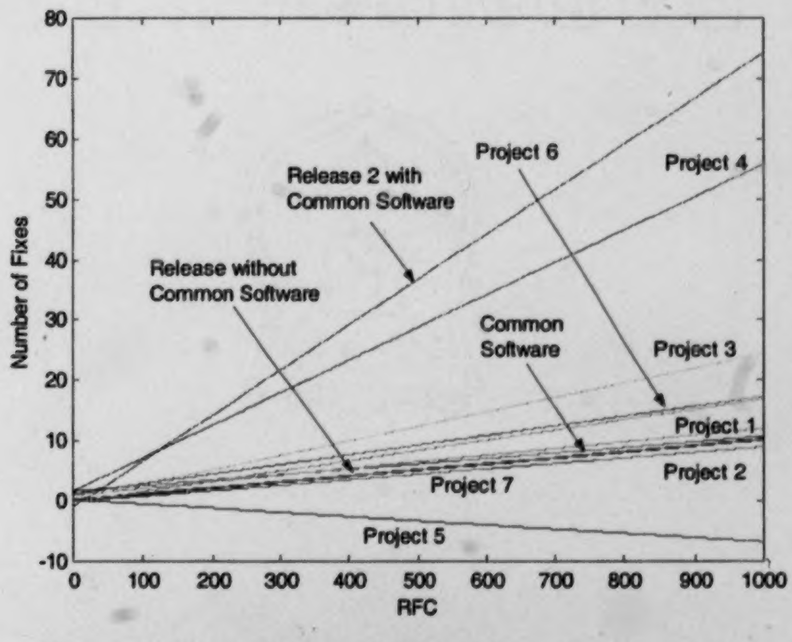

Figure 19: Regression model obtained from the second release of telecommunication projects

Several conclusions can be drawn from the undertaken experiments:

1. Slope of the linear models can be determined quite precisely and with high confidence.
2. Intersections (i.e. values of dependent variable when independent variables assume zero value) differ significantly, which means that such models need to be recalibrated for each data set.
3. Inclusion of the Common Software part in this model lowers the confidence of the obtained model.

The performance of this method will be compared to the model that optimizes regression error on data and model level.

This approach has two main shortcomings. First, extending this model to nonlinear regression models is in no way straightforward. This presents a major limitation of the method, since software data usually cannot meet all the requirements for applying linear regression. Second, the time dimension is needed for this type of analysis. Even though exact matching in time is not necessary, this information may still be unavailable or irrelevant in some software projects.

Another method introduced in this study, with the same goal of finding optimal general regression model is minimizing the error on the level of data and model simultaneously.

The results are also based on univariate models with RFC as a predictor, so that the comparison of the proposed methods can be performed.

The procedure as explained in the background section is closely followed. The parameter that can be varied across experiments is the coefficient γ that should balance two error components. The analysis showed that this factor is not significant, meaning that even when the range for γ is very wide, the results do not change significantly. In this step, the models are devised using the whole data sets (i.e. no validation is present), and they are shown in Table 28.

	Regression model	Performance Index
First release	Y = 0.009*X + 1.183	Q = 1951.31
Second release	Y = -0.009*X + 0.914	Q = 1641.71

Table 28: Regression models based on training and testing approach for telecom data

In order to quantify the robustness of the derived models, the classical training and testing approach is chosen. The data is split into 60% - 40% and the sum of square errors is adopted as the indication of performance. The same approach is applied to results from both methods. Since the training and testing subsamples are of different sizes, the performance index for the training part would naturally be higher. Therefore, the index is normalized with respect to the size of the training part in order to enable their proper comparison.

		Model	Performance index
SURE	First release	Y = 0.009*X + 1.696	Q = 2002.56
	Second release	Y = 0.010*X + 0.105	Q = 1551.39
Data-model	First release	Y = 0.009*X + 1.183	Q = 1951.31
	Second release	Y = -0.009*X + 0.914	Q = 1641.71

Table 29: Comparison of the two methods for combining regression models

The validation procedure indicates that in the first release data-model performs slightly better than SURE, but in the case of the second release the results are just the opposite. Therefore, one more approach of validating the models is applied.

One more idea of how to validate results obtained from the experiments is to devise regression models from the first release and evaluate their performance on the subsequent release. This is strategically important in industry, and if successful it would mean that such regression model possesses good predictive ability.

Examples of such validation are known in literature. Briand *et al.* (2000a) investigate two mid-size Java systems and attempt to build prediction models on one product and then apply it to subsequent software development projects. The results suggest that the ranking of classes according to their fault proneness could be applied to other software products, but that the process is in no way straightforward.

The performances of the proposed regression models on two subsequent releases of data used in this study are shown in Table 30.

In this case, data-model method performs slightly better than SURE method, although the results are still similar.

	Training (first release)	**Testing (second release)**
SURE	$Y = 0.009*X + 1.696$ $Q = 2002.560$	$Q = 1548.37$
Data-model	$Y = 0.009*X + 1.183$ $Q = 1953.31$	$Q = 1400.91$

Table 30: Validation of results across two releases of telecom projects

The results do not provide clear indication as to the model that should be the most appropriate in all cases. In general, the performance of all the investigated models is comparable. SURE model is promising because it can deal with the problem of correlated errors. However, it does not outperform the other model in general probably because the existing correlations are not that high. Data-model approach certainly adds one more dimension in looking at this problem and is more flexible in terms of extending it to other types of regression models (e.g. nonlinear).

5.4 Application of decision trees to software data

Decision tree learning has already been presented earlier in the study. Since classical regression models are a more formal way of modeling data, and considering the challenging characteristics of software data, decision trees could be a promising way to model software data. Therefore, this research aims at applying decision trees to software data in order to gain insight into complementary structures of software processes compared to the one offered by statistical modeling.

The classification method of building decision trees based on software data is applied in this research for two reasons:

1. To identify the best among available predictors
2. To extract rules describing software behavior of interest

Two analyses are undertaken to achieve these goals.

First, this study is concerned with finding rules in order to describe the software behavior using software metrics as classifiers. In this case, classifiers are object-oriented software metrics from the CK metrics suite. For this purpose, the number of revisions known for each class is dichotomized, i.e. transformed into a binary number. Decision trees for different combinations of attributes and different sizes are grown, and their correctness and usability are assessed.

The data sets are quite small in general; therefore the largest data set from both releases is chosen to demonstrate the process. This approach is taken because deriving rules based on small data sets would present a serious threat to validity of the extracted rules.

Since the data under investigation shows high presence of zeros for output values, and classical statistical approaches do not deal with this problem very well, it is assumed that categorizing the data in only two groups would be the least strict. Decision trees are grown with S-PLUS statistical package.

The approach taken is selecting the subsets of data sets and performing the testing and training. The algorithm is quite straightforward:

1. A subset of given data, called the "training set" is selected randomly as 60% of the total size
2. Pruning is applied with the chosen sizes from 2 up to the unrestricted size
3. Trees are grown for all the sizes
4. Trees are tested using testing set and the success of classification is computed
5. The results are observed and discussed
6. Rules are extracted with respect to performance in both training and testing

The results of this experiment are shown in Table 31.

According to the success rate, the performance increases in both training and testing with the tree size. These results were unexpected, since overfitting the data should not produce good results. These results indicate that in the resulting trees there are both nodes that perform very well and those that do not. When they are averaged, the outcomes are quite similar.

Tree size			Number of data points				Success rate (%)	
Desired	**Returned**	**Actual**	**Train. missed**	**Train. hits**	**Test missed**	**Test hits**	**Training**	**Testing**
2	2	2	59	89	45	54	60.14	54.55
3	3	3	51	97	42	57	65.54	57.58
11	11	10	39	109	42	57	73.65	57.58
14	14	13	34	114	42	57	77.03	57.58
15	15	14	34	114	39	60	77.03	60.61
19	19	16	34	114	40	59	77.03	59.60
20	20	17	31	117	35	64	79.05	64.65
23	23	19	29	119	32	67	82.00	67.00

Table 31: Results of comparative experiment with different tree sizes

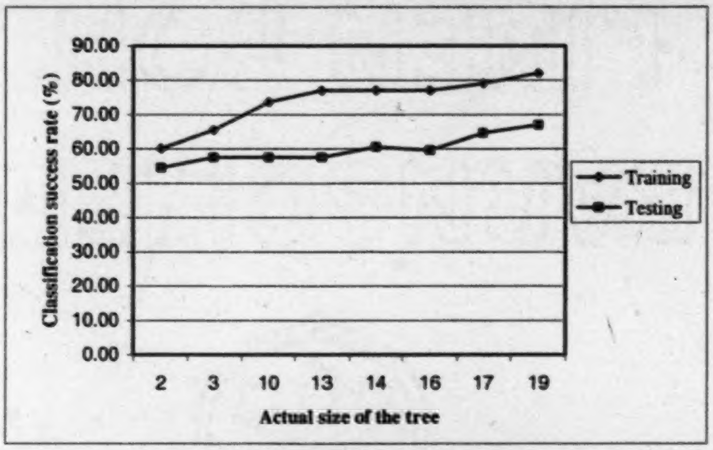

Figure 20: Graphical representation of classification success for both training and testing data

In order to refine the results, the next step was to determine which portion of data ends in each node of the tree and what is the performance of each node separately. The nodes with high performance are those from which good rules could be extracted.

In order to explore each node separately and for different tree sizes, experiments need to be performed across these two dimensions.

Figure 21 shows the full decision tree generated using S-PLUS for the chosen application.

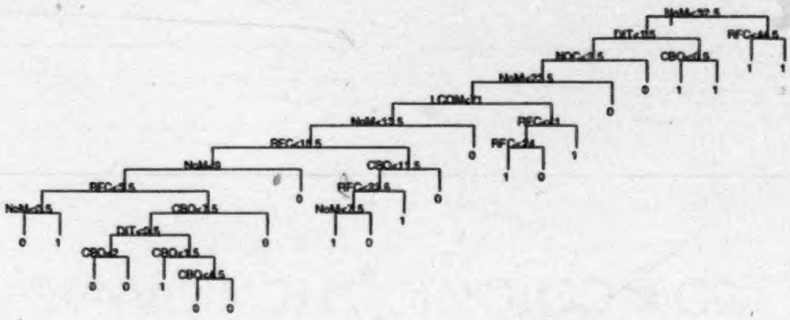

Figure 21: Full size tree generated with S-PLUS

This figure also illustrates the following characteristics of S-PLUS tree generation procedure.

First, although the returned size is 23, this tree actually has 18 terminal nodes.

The overview of desired sizes and actual sizes for this data set is presented in Table 32.

Row #	Actual size	Desired size
1	2	2
2	3	3,4
3	10	5,6,7,8,9,10,11
4	13	12,13,14
5	14	15
6	16	16,17,18,19
7	17	20
8	18	21,22,23

Table 32: Overview of tree sizes generated with S-PLUS for the telecom data set

The results in Table 31 show that although the size of the tree is specified in the pruning process, S-PLUS cannot always generate the tree of that particular size. Even if such a tree can be generated, some of the splits may not represent real splits, as explained earlier. This can be explained on the example of row number 4, where the specified size in the pruning process was specified from 5 up to 11, and still the resulting tree was always of the actual size 10. Therefore, the information of both sizes is recorded in order to have the detailed information on the whole experiment.

Second, all metrics used for this study assume values on absolute scale, meaning that values like 5.5 should not appear. When the rules are extracted, these values are adjusted to the closest lower or higher integer, as explained in the background section.

The experiment is run for all the actual tree sizes, and for each node the number of data points ending in it is calculated. A sample result in the form of histograms is shown in the following figure. The first bar of each node represents non-faulty class and the second faulty ones.

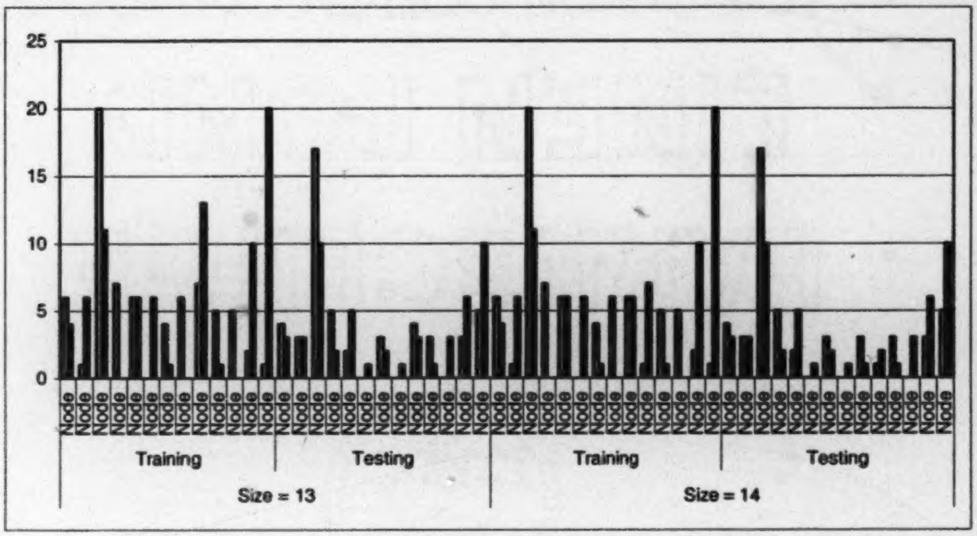

Figure 22: Sample histogram for tree sizes 13 and 14 generated with S-PLUS

Histograms in Figure 22 represent how many data points in the training and testing parts end up in each node of the tree. Two values are present in the output, zeros for non-faulty and ones for faulty classes. The number on the y axis shows how many data points reached the node, the first bar always represents how many classes were not faulty and the second how many were faulty.

In the process of selecting the "best" nodes (or rules), the aim is to identify those where one group prevails significantly over the other. It means that most of the data points give one outcome. Such nodes are selected and tested because S-PLUS does not provide automatic testing of trees, thus making this process very time-consuming. **Error!**

Reference source not found. shows the most promising rules in terms of performance, as obtained with S-PLUS.

However, when assessing the overall performance of rules two aspects should be considered. First, the success of classification is a strong indication of the quality of rules. Second, the number of data points ending in each node should be taken into account. This means that not only the nodes with high success rate should be considered. The part of data that confirms such a rule should also be taken into account. This is explained on the examples of following rules.

Rule #	Rule	No. of points		Success Rate (%)	
		Training	Testing	Training	Testing
1	If NOM>32 => FAULTY CLASS	1 zero 20 ones	5 zeros 10 ones	95.2%	66.7%
2	If NOM>32 and DIT>1 => FAULTY CLASS	2 zeros 10 ones	3 zeros 6 ones	83.3%	66.7%
3	If NOM in (24,32) and DIT<2 and NOC<3 =>NON-FAULTY CLASS	5 zeros 1 one	3 zeros 1 one	83.3%	75%
4	If NOM>13 and DIT<2 and NOC<3 and LCOM<71 and RFC>22 => FAULTY CLASS	0 zeros 6 ones	0 zeros 1 one	100%	100%
5	If NOM in (9,13) and DIT<2 and NOC<3 and LCOM<71 and RFC<16 => NON-FAULTY CLASS	7 zeros 0 ones	5 zeros 2 ones	100%	71.4%
6	If NOM<24 and DIT<2 and NOC<3 and LCOM>71 and RFC>41 => FAULTY CLASS	1 zero 7 ones	1 zeros 2 ones	87.5%	66.7%
7	If NOM<14 and DIT<2 and NOC<3 and LCOM<71 and RFC>15 and CBO>7 => NON-FAULTY CLASS	5 zeros 0 ones	1 zero 0 ones	100%	100%
8	If NOM<9 and DIT=1 and NOC<3 and LCOM<71 and RFC in (4,15) and CBO<2 => FAULTY CLASS	1 zero 4 ones	1 zero 5 ones	80%	83.3%

Table 33: Success rates of extracted "best" rules from the telecom data set obtained using S-PLUS

Rules 4 and 7 are ideal in terms of performance, since the classification rates for both training and testing for these two rules are 100%. However, the fact that the success of this result is obtained on a comparatively small portion of data points should not be disregarded; therefore the probability of getting such high results by chance is not negligible. On the other hand, rule number 2 has somewhat lower success rate (83.3% on training and 66.7% bon testing data).

Clearly, both aspects should be considered simultaneously in order to draw credible results on which rules really represent the phenomena of interest. This study recognizes both aspects and attempts to identify those rules that meet maximum of these two at the same time.

In that sense, the following rules are selected as best representatives:

- If NOM>32 => FAULTY CLASS
- If NOM>32 and DIT>1 => FAULTY CLASS

These two rules have quite high classification rates while covering the most data when compared to other rules. Some further discussion on how the results can be compared is provided after the results of the second experiment.

This study also considers the problem of collinearity, which has been proven to exist among some measures in the CK suite. The same telecom data is used as in the meta-analytic study on collinearity. The goal of this investigation is twofold:

1. To avoid incorrect final rules
2. To reduce the length of the extracted rules

The first issue is analogous to the concern raised in many studies and discussed in this thesis that inclusion of collinear metrics results in ambiguous statistical models. Therefore, this part is focused on determining whether that approach should be adopted in the decision tree learning process.

The second issue is of a quite practical nature. If all but one collinear metric can be excluded it would result in more compact rules. The collinearity issue is investigated on the same data used in the previous experiment, and two other artificially generated data sets, in order to confirm or disconfirm the obtained results.

From Figure 21 it is clear that all three collinear metrics, RFC, CBO and NOM appear in the full-sized tree, implying that decision tree does not recognize this fact. Therefore, the same problem is examined on two simulated data sets, both of size 500 when metrics are generated using the uniform distribution function, and the other with skewed distribution, in order to simulate the characteristics of the telecom data sets.

One of the collinear measures, NOM, is generated freely while the other two are calculated using arbitrary linear functions.

The results are as follows:

1. In the uniformly distributed data measure that simulates RFC does not appear, CBO appears only once, and the remaining splits belong to the analogue of NOM.
2. In the second data set, analogue of RFC again does not appear, CBO appears six times and the remaining classifiers are attributed to the analogue of NOM.

The conclusion from these experiments can be drawn that if the strong linear dependence in the data really exists, the decision tree will recognize this and eventually discard most of them. Therefore, it is not necessary to worry upfront about the presence of collinearity and excluding in order to preserve the correctness of the derived models.

The second experiment on the application of decision trees is undertaken on the NASA Ames software project data.

NASA data set provides information on software problem reports and is analyzed in this part of the research to create rules that would predict the time needed to fix the problems. The measures used are related to the number of modules to be changed, phase in the software development process where the problem occurred and the complexity of the problem. The data is preprocessed as described in Section 3. Results from trees generated with S-PLUS are shown in Table 34.

The results indicate that the simplest rule (number 1) is the best with respect to two criteria:

- Performance over training and testing parts is consistent
- Results are significant since 90.25% of the data is covered by this rule

Number of Modules To Change is clearly a very strong indicator of the time invested in fixing a software problem report. Therefore, even though this rule is very significant, some more rules should also be considered. The following rules are considered to represent the most successful rules:

- If Modules To Change < 3 and Error Type in (Computation, Logic, Data I/O) => Less than 1 hour
- If Modules To Change = 3 and Error Type = Computation=> Less than 1 hour

- If Modules To Change > 3 and Error Type in (S/W Interface, Data Handling, Database, Improper Initialization, Hardware Logic, Data Output, Data Definition, Other) and Origin in (Coding, Previous Problem, etc.) => Less than 1 hour

Rule #	Rule	No. of data points ending in the rule		Classification rate	
		Training	Testing	Training (%)	Testing (%)
1	**If Modules To Change < 3 => Less than 1 hour**	1312	881	**90.5%**	**90.1%**
2	If Modules To Change = 3 => Less than 1 hour	70	46	65.7%	56.5%
3	If Modules To Change > 3 => More than 1 hour	76	45	64.5%	55.6%
4	**If Modules To Change < 3 and Error Type in (Computation, Logic, Data I/O) => Less than 1 hour**	50	23	**76.0%**	**60.9%**
5	If Modules To Change > 3 and Error Type in (S/W Interface, Data Handling, Database, Improper Initialization, Hardware Logic, Data Output, Data Definition, Other) => Less than 1 hour	26	22	57.7%	50.0%
6	**If Modules To Change = 3 and Error Type = Computation => Less than 1 hour**	4	3	**75.0%**	**100%**
7	If Modules To Change = 3 and Error Type in (S/W Interface, Data Handling, Database, Improper Initialization, Hardware Logic, Data Output, Data Definition, Other) => More than 1 hour	66	44	34.8%	45.5%
8	If Modules To Change > 3 and Error Type in (S/W Interface, Data Handling, Database, Improper Initialization, Hardware Logic, Data Output, Data Definition, Other) and Origin in (Revision Request, Requirements, Design) => More than 1 hour	12	7	58.3%	71.4%
9	**If Modules To Change > 3 and Error Type in (S/W Interface, Data Handling, Database, Improper Initialization, Hardware Logic, Data Output, Data Definition, Other) and Origin in (Coding, Previous Problem, etc.) => Less than 1 hour**	14	15	**71.4%**	**60.0%**
10	If Modules To Change = 3 and Origin = Design and Error Type in (Computation, Logic) => Less than 1 hour	23	16	56.5%	43.8%
11	If Modules To Change = 3 and Origin = Design and Error Type in (S/W Interface, Data Handling, Database, Improper Initialization, Hardware Logic, Data Output, Data Definition, Other) => Less than 1 hour	7	7	71.4%	42.9%
12	If Modules To Change = 4 and Error Type in (Logic, Data I/O) and Origin in (Revision Request, Requirements) = > More than 1 hour	5	2	60.0%	0.0%
13	If Modules To Change and Error Type in (Logic, Data I/O) and Origin in (Revision Request, Requirements) => Less than 1 hour	6	1	50.0%	100%

Table 34: Summary of NASA Ames analysis as obtained with S-PLUS

In order to gain deeper understanding of decision tree learning process, the same data set is used for extracting rules using CART software. The main advantage of this software package is that it grows decision trees having in mind the testing data set. That alleviates problems related to overfitting, since it instantly tests the grown tree and chooses the optimal one, based on the selected criterion.

			Learning						Testing					
			Class 1			Class 2			Class 1			Class 2		
Tree	Number Of Nodes	Node	Cases in Node	Cases Class 1	% of Node Class 1	Cases in Node	Cases Class 2	% of Node Class 2	Cases in Node	Cases Class 1	% of Node Class 1	Cases in Node	Cases Class 2	% of Node Class 2
1	2	1	1118	1032	**92.308**	340	111	**32.647**	746	685	**91.821**	226	71	**31.423**
		2	340	229	**67.353**	1118	86	**7.692**	226	155	**68.584**	746	61	**8.182**
2	3	1	1313	1189	**90.556**	75	49	**65.333**	886	796	**89.842**	40	22	**55.000**
		2	70	46	**65.714**	70	24	**34.286**	46	26	**56.522**	46	20	**43.478**
		3	75	26	**34.667**	1313	124	**9.444**	40	18	**45.000**	886	90	**10.158**
3	2	1	1298	1178	**90.755**	160	77	**48.125**	878	791	**90.091**	94	45	**47.872**
		2	160	83	**51.875**	1298	120	**9.245**	94	49	**52.128**	878	87	**9.909**
4	4	1	616	590	**95.779**	145	73	**50.345**	419	398	**94.988**	86	42	**48.837**
		2	502	442	**88.048**	195	38	**19.487**	327	287	**87.768**	140	29	**20.714**
		3	195	157	**80.513**	502	60	**11.952**	140	111	**79.286**	327	40	**12.232**
		4	145	72	**49.655**	616	26	**4.221**	86	44	**51.163**	419	21	**5.012**
5	2	1	1313	1189	**90.556**	145	73	**50.345**	886	796	**89.842**	86	42	**48.837**
		2	145	72	**49.655**	1313	124	**9.444**	86	44	**51.163**	886	90	**10.158**
6	4	1	699	664	**94.993**	160	77	**48.125**	484	450	**92.975**	94	45	**47.872**
		2	502	442	**88.048**	97	25	**25.773**	327	287	**87.768**	67	13	**19.403**
		3	97	72	**74.227**	502	60	**11.952**	67	54	**80.597**	327	40	**12.232**
		4	160	83	**51.875**	699	35	**5.007**	94	49	**52.128**	484	34	**7.025**
7	3	1	616	590	**95.779**	340	111	**32.647**	419	398	**94.988**	419	398	**94.988**
		2	502	442	**88.048**	502	60	**11.952**	327	287	**87.768**	327	287	**87.768**
		3	340	229	**67.353**	616	26	**4.221**	226	155	**65.584**	226	155	**68.584**

Table 35: Comparison of performance of the best trees produced by CART

A series of experiments with CART was performed, varying the parameters explained previously.

Table 35 provides the performance of seven best trees produced by CART. It is evident that CART generates much smaller trees, because the problem of overfitting is partially overcome. The results are presented in the similar fashion for both S-PLUS and CART trees, to enable proper comparison.

Moreover, in CART analysis it is possible to compare the performance of whole trees, in addition to comparing separate rules. However, the important issue is how to actually evaluate the obtained results. The main dilemma is how to compare the performance of trees and two issues arise here:

1. In case of trees of different sizes, is it better to obtain fewer nodes that cover larger portions of data or it is preferable to have more nodes that are purer but cover fewer data?
2. In case of trees of the same size, should more importance be given to the training or testing part of the data?

The following questions are discussed in more details on actual examples.

For the first question, trees number 1 and 4 are taken as examples. The success rates for these two trees are shown in Table 36.

			Learning		Testing	
Tree	Number of nodes	Node	% of Node Class 1	% of Node Class 2	% of Node Class 1	% of Node Class 2
1	2	1	92.31	32.65	91.82	31.42
		2	67.35	7.69	68.58	8.18
4	4	1	95.78	50.34	94.99	48.84
		2	88.05	19.49	87.77	20.71
		3	80.51	11.95	79.29	12.23
		4	49.65	4.22	51.16	5.01

Table 36: Comparison of two different-sized trees

The question is how to compare these two results in terms of performance. The first tree (of size 2) gives quite pure first node while the second node has lower success rate, especially on the testing data. On the other hand, the second tree has the first two nodes of similar even higher performance compared to the first tree. Clearly, fewer data points end up in these nodes, meaning that the statistical significance is lower in this case.

The second question is discussed on the examples of trees 1 and 5. Table 37 presents the success rates for these two trees.

			Learning		Testing	
Tree	Number of nodes	Node	% of Node Class 1	% of Node Class 2	% of Node Class 1	% of Node Class 2
1	2	1	92.31	32.65	91.82	31.42
		2	67.35	7.69	68.58	8.18
5	2	1	90.56	50.34	89.84	48.84
		2	49.65	9.44	51.16	10.16

Table 37: Comparison of the same-sized trees

Tree number 1 has slightly higher success rate for class 1 than tree 5. However, the classification of tree 1 for the second class is poorer than in case of tree 5. At the same time, the consistency of performance for both trees on training and testing data is good. The decision needs to be made at this point whether the primary goal is to achieve purer nodes and extract rules from them in order to get high-performing rules, or covering as much of the data as possible is more important. The solution of how to choose the best rule(s) is not straightforward. It depends on the goals set for each specific experiment using decision trees, and probably depends on the application domain, as well.

Having in mind this discussion, the following rules are identified as "best" as obtained in CART analysis:

- If Modules To Change in {1,11,12,...} => Time to fix SPR is less than 1 hour
- If Modules To Change = 3 => Time to fix SPR is more than 1 hour
- If Modules To Change in {1,11,12,...} and SPR Origin in {Coding, Previous SPR, Other, Comments in Source Code} => Time to fix SPR is less than 1 hour
- If Modules To Change in {0,2} => Time to fix SPR is less than 1 hour

The results from S-PLUS and CART show that the rules somewhat differ, although parts overlap. The following rules represent a union of the results from both analyses:

- In both analyses Number of Modules To Change (NMC) is the strongest classifiers
- If NMC is less than 3, usually it takes less than 1 hour to fix the problem
- If NMC is equal 3, then additional parameters (such as the type of the problem) determine the time to fix it
- CART finds that there are cases where even for NMC higher than 11, the time to fix the problem is shorter than 1 hour – this is not confirmed in S-PLUS
- The least difficult problems to solve are problems remaining from some previous problem reports, coding, computation and logic issues

The analysis presented here still shows comparable results in all discussed cases, and taking any of the rules would lead to decent prognosis.

In conclusion, when the results of the best-performing trees and rules obtained using S-PLUS and CART are compared, an overlap exists in small-size trees. For larger trees it is difficult to compare since CART does not produce overly large trees.

5.5 Software inspection experiment

The goal and the description of the procedure for meta-analyzing the results from software inspection experiments are described earlier in this study. The results from the meta-analysis together with the discussion on the homogeneity of the data are presented as follows.

Since the goal of this part of research is to compare the effectiveness of defect detection of two reading techniques, namely *Ad-hoc* and *Perspective-Based Reading*, the data from each project is divided in two parts with respect to the technique used.

Following the procedure explained, the first measures for this experiment are mean and standard deviation of each project separately. These statistics are calculated for the discovered number of defects, since that measure reflect the success of the technique to identify software faults. Since the projects used in this experiment are of different sizes, ranging from 12 to 15 records, the data is normalized. Table 38 summarizes descriptive statistics for all applications and each technique separately.

		Application 1		**Application 2**		**Application 3**		**Application 4**	
Ad-hoc	St. dev. 1	1.60	1.50	3.60	3.30	3.30	4.80	3.20	3.20
PBR	St. dev. 2	0.98	2.10	2.20	3.10	3.60	4.20	5.20	2.90
	Pooled St. dev.	1.33	1.82	2.98	3.20	3.45	4.51	4.32	3.05
Ad-hoc	Mean 1	3.20	1.30	7.90	9.50	8.90	7.30	9.80	9.50
PBR	Mean 2	1.830	2.50	5.80	6.80	5.90	7.80	9.30	6.20
Ad-hoc	Mean 1 – normalized	4.98	2.43	6.91	9.50	16.61	13.63	9.80	9.85
PBR	Mean 2 - normalized	2.85	4.67	5.07	6.80	11.01	14.56	9.30	6.43

Table 38: Descriptive statistics for each application and each reading technique

The data on defects are based on the "seeded" defects, which are intentionally introduced into documents. Since the source of data is four applications, different number of defects is injected in each. These are presented in Table 39.

	Project 1				**Project 2**			
	App.1	**App.2**	**App.3**	**App.4**	**App.1**	**App.2**	**App.3**	**App. 4**
Defects	18	15	32	28	15	15	28	27

Table 39: Summary of number of seeded defects for each project and application

The size of each study together with the portions of data belonging to each technique, and the bias correction factor are shown in Table 40.

	Project 1				**Project 2**			
	App.1	**App.2**	**App.3**	**App.4**	**App.1**	**App.2**	**App.3**	**App. 4**
Ad-hoc	6	6	6	6	6	7	8	6
PBR	6	6	6	6	8	6	6	8
Total	12	12	12	12	14	13	14	14
Correction factor – J(N-2)	0.9228	0.9228	0.9228	0.9228	0.9359	0.9300	0.9359	0.9359

Table 40: Study sizes and the corresponding bias correction factors

The following step is to calculate unbiased effect sizes and their weights that reflect their contribution in the overall effect size. The results are shown in Table 41.

	Project 1				**Project 2**			
	App.1	**App.2**	**App.3**	**App.4**	**App.1**	**App.2**	**App.3**	**App. 4**
Individual effect size (g_i)	1.48	0.61	0.65	0.78	0.81	-0.10	0.11	1.01
Weights	0.12	0.12	0.12	0.12	0.13	0.13	0.13	0.13

Table 41: Individual effect sizes and their weights

The final pooled effect sized based on this meta-analytic technique is

$$g = 0.517$$

Since this study adopts 0.5 effect size as a result of practical significance, this can be considered as a valid result.

In order to confirm the validity of this meta-analytic technique, the test of homogeneity is performed. Following the procedure provided earlier, Q value obtained is

$$Q = 0.115$$

This value needs to be compared with the 100(1-α) of chi-square value for k-1 degrees of freedom, which in this case is 0.147.

Since the Q value is lower than the chi-square test value, it can be concluded that this meta-analysis is valid, and that Perspective-Based reading is in general more effective for defect detection in software documents than ad-hoc reading technique.

6. Conclusions and recommendations

Software engineering involves the study of the means for producing high-quality software products with predictable costs and schedules. Due to the growing diffusion of complex software systems, industries are looking for metrics and models capable of producing accurate quantitative predictions of software behavior.

The purpose of this study is to explore quantitative software engineering methods that would enable better knowledge management in software engineering. The focus is mainly on object-oriented software systems due to their extensive presence and importance in the software market.

This study attempts to broaden the application of statistical regression for modeling software data in terms of combining regression models across different projects and environments. For this purpose, two techniques are proposed and investigated: Seemingly Unrelated Regression (SURE) models and optimizing the model at data and model level. The results of the two proposed techniques are assessed based on the performance criteria.

Each of the proposed techniques shows some advantages. SURE model deals well with the projects when they share some common characteristics. On the other hand, optimizing the regression error on data and model level simultaneously is a novel approach embracing the two important aspects, while not requiring a time stamp in the data and being easily extensible to any type of regression models. The following conclusions can be drawn:

- The outcomes are not very strong in favor of one specific technique in general. Each of the models shows good performance on some data sets, implying that the derived models are still context-dependent.
- Further investigation of both models should be performed, preferably on higher number of data sets with different characteristics and of high quality.

In order to apply statistical models in a correct way, exact characteristics of the applied software measures should be known. In particular, this research investigates the presence of collinearity among three measures in the CK object-oriented metrics suite, CBO, NOM and RFC and narrow ranges on absolute scale for DIT and NOC.

Collinearity represents the situation when metrics describe the same software attributes. To confirm or disconfirm collinearity among the mentioned metrics, the meta-analytic technique of Weighted Estimators of a Common Correlation, based on correlations between software measures, is applied. The results indicate that there is indeed high collinearity between the three metrics from the suite. The achieved results are significant for the following reasons:

- The analysis shows that only one of these measures is useful while other two are redundant. Knowing that software metrics extraction can be a time- and effort-consuming process, this can lead to considerable time savings.
- When certain multivariate statistical models, such as linear and log-linear, are used to model software data using collinear variables, the prediction models thus created can be either ambiguous or incorrect. Hence, it is recommended that when the collinearity is proven to exist, all but one metric should be removed from the prediction model. However, it should be emphasized that there are models and techniques for which collinearity among predictor variables does not present a threat, and the validity of such models is not in question even if the collinearity is present.

The analysis is undertaken to both public domain and industrial projects with similar results, proving the confidence in the achieved result.

Another aspect studied is the narrow range of two metrics from the CK suite, NOC and DIT. Statistical sign test is applied in order to investigate this aspect of object-oriented metrics. The results confirm that in all cases the metrics are below the initially chosen threshold 10. The study goes further to determine if in specific cases lower thresholds could be confirmed, resulting in the lowest maximum value of 4 for NOC in Java projects.

Two important conclusions can be drawn from this study:

- Current practice in object-oriented systems is quantified showing to what extent the concepts of this methodology are used in software projects.
- When statistical models are used to describe the behavior of software systems, careful choice of the models needs to be made to deal with such variables.

Meta-analysis is also applied to identify those software attributes that describe software behavior in the best way. To achieve this, the relationship between internal (CK metrics) and external metrics (number of defects or alternatively revisions) is meta-analyzed. The purpose of this experiment is to identify the measures that are consistently highly correlated with the number of defects, implying that they can serve as good indicators of fault-proneness of software modules.

The investigation is carried out on two consecutive releases of industrial data, and the conclusions are as follows:

- Some of the measures (LOC, RFC, CBO and NOM) show higher correlations with respect to number of revisions for a class
- These correlations are not significantly higher compared to the remaining CK measures

These conclusions imply that, based on the data used in this study, univariate statistical models are not likely to produce optimal results in building prediction systems, that is, there is no single measure that can be used to describe the defect-proneness of classes alone.

Although the test of homogeneity in this analysis formally confirms the validity of meta-analytic approach, it is still recommended to carefully investigate the characteristics of each data set before performing meta-analysis. Special attention should be paid to the difference in data distribution, since the introduction of the Common software part of code in this meta-analysis led to lower correlations and higher heterogeneity in data. Considering that the data sets across projects differ substantially, this issue should be investigated further.

Another way to approach the issue of software defect-proneness is to apply software inspection process for early identification of software problems. This research attempts to compare and combine data on effectiveness of two reading techniques for software inspection in order to draw a conclusion which one is more effective. The meta-analysis is undertaken on eight data sets and the results prove that Perspective-based reading is a more effective reading technique for discovering defects in software documents than Ad-hoc reading. The pooled effect size is higher than 0.5 therefore presenting the result of practical significance. In addition, the homogeneity test is satisfied, thus proving that it is possible to combine the results across the available experiments.

A nonparametric method of building decision trees based on software data is applied to two industrial data sets coming from different environments. The created decision trees are used to extract rules that explain what and to what extent describes software projects. Two different commercial tools, the statistical package S-PLUS and CART are used in order to apply the technique of decision tree learning to the available software projects. This research provides the discussion of the important aspects that need to be taken into account when assessing the overall success and usefulness of the extracted rules. Two analyses are undertaken with the following goals:

- For the first project, the goal was to derive rules identifying faulty object-oriented classes based on CK measures using S-PLUS software. The rules devised are discussed regarding the performance on both training and testing data sets; the best performing rules are highlighted.
- The second data set is used to extract rules describing the time needed to fix software problem reports in NASA avionics project based on measures collected during the project. The analysis is undertaken using both S-PLUS and CART and the rules extracted using both tools are compared. The comparison of the rules shows that there is a considerable overlap in the extracted rules.

The analysis indicates that although there are minor limitations, useful conclusions can be drawn using this classification technique.

There is a large amount of research to be done in terms of studying empirically software development. Software metrics research and practice has helped to build up an empirical basis for software engineering, which is an important achievement. It is clear that software engineering researchers and practitioners need to develop more valid metrics and use them to record and collect more detailed information about the software projects, the people involved, and the development environment, so that the more plausible models of software processes can be specified, tested, and validated. This will ultimately lead to better software management practices.

The results and the inferences of this study are still limited by the small sample size of individual data set, heterogeneous project data, and uncertain data quality. To alleviate these problems and achieve more credible results, empirical research should be performed with as many industrial systems of varying application domains as possible, supported by well-designed hypothesis. The work performed in this study can be continued using new available datasets and applying the framework for empirical investigation established in this study.

7. References

Albrecht, A.J., J. Gaffney (1983) "Software function, source lines of code and development effort prediction," *IEEE Transactions on Software Engineering*, *9*(6): 639-648

Aron, A., E.N. Aron (1997) Statistics for the behavioral and social sciences, Prentice Hall

Baker, A.L., J.M. Bieman, N. Fenton, D.A. Gustavson, A. Melton, R. Whitty (1990) "A Philosophy for Software Measurement," *The Journal of Systems and Software*, *12*: 277-281

Bansiya, J., C. Davis (1999) "Design and code complexity metrics for OO classes," *Journal of Object Oriented Programming*, *12*(1): 35-40

Barnston, A.G. (1994) "Linear statistical short-term climate predictive skill in the Northern Hemisphere," *Journal of Climate*, **7**: 1513-1564

Basili, V.R., H.D Rombach (1988) "The TAME project: Towards improvement-oriented software environments," *IEEE Transactions on Software Engineering*, *14*(6): 758-773, *October*

Basili, V.R., L.C. Briand, W.L. Melo (1996) "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, *22*(10): 751-761

Belanger, S.E. (1997) "Literature Review and Analysis of Biological Complexity in Model Stream Ecosystems: Influence of Size and Experimental Design," *Ecotoxicology and Environmental Safety*, **36**: 1-16

Belsley, D.A. (1991) Conditioning diagnostics: Collinearity and Weak Data in Regression, J. Wiley, New York

Bevan, N., M. Macloed (1993) "Usability Assessment and Measurement," *UNICOM SEMINARS*, Middlesex, UK, pp. 167-192

Bieman, J.M., N.E. Fenton, D.A. Gustafson, A. Melton, R. Whitty (1992) "Moving from Philosophy to Practice in Software Measurement," in: Denvir et. al.: *Formal Aspects of Measurement, Springer Verlag*, pp. 38-59

Booch, G. (1994) Object-Oriented Analysis and Design With Applications, Object Technology Series, Addison-Wesley

Bourque, P., V. Cot'e, (1990) "An Experiment in Software Sizing with Structured Analysis Metrics," *The Journal of Systems and Software*, ***12***: 159-172

Breiman, L., J. Friedman, R. Olshen, C.J. Stone (1984) Classification and regression trees, Belmont, California: Wadsworth International Group

Briand, L.C., K. El Emam, S. Morasca (1996) "On the Application of Measurement Theory in Software Engineering," *Journal of Empirical Software Engineering*, ***1***(1): 61-88

Briand, L., K. El Emam, O. Laitenberger, T. Fussbroich (1997) " Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects," *International Software Engineering Network Technical Report, ISERN-1997-21*

Briand, J.W., S.V. Ikonomovski, H. Lounis (1999a) "Investigating Quality Factors in Object-Oriented Designs: an Industrial Case Study," *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, May 16-22, pp.345-354

Briand, L.C., S. Morasca, V.R. Basili (1999b) "Defining and Validating Measures for Object-Based High-Level Design," *IEEE Transactions on Software Engineering*, ***25***(5): 722-743, *September/October*

Briand, L.C., J. Wüst (1999c) "The impact of Design on Development Cost in Object-Oriented Systems," *International Software Engineering Network Technical Report, ISERN-99-16*

Briand, L., W. Melo, J. Wüst (2000a) "Assessing the Applicability of Fault-Proneness Models Across Object-Oriented Software Projects," *International Software Engineering Network Technical Report, ISERN-00-06*

Briand, L.C., J. Wüst, J.W. Daly, D.V. Porter (2000b) "Exploring the relationships between design measures and software quality in object-oriented systems," *The Journal of Systems and Software*, ***51***: 245-273

Briand, L.C., J. Wüst (2001) "Modeling Development Effort in Object-Oriented Systems Using Design Properties," *IEEE Transactions on Software Engineering, 27*(11): 963-986, *November*

Brooks, F.P. (1987) "No Silver Bullet - Essence and Accidents of Software Engineering," *Computer Magazine, April*

Bruegge, B., A.H. Dutoit (2000) Object-oriented software engineering: conquering complex and changing systems, Upper Saddle River, NJ: Prentice Hall

Cartwright, M., M. Shepperd (2000) "An Empirical Investigation of an Object-Oriented Software System," *IEEE Transactions on Software Engineering, 26*(8): 786-796, *August*

Chernak, Y. (1996) "A Statistical Approach to the Inspection Checklist Formal Synthesis and Improvement," *IEEE Transactions on Software Engineering, 22*(12): 866-874, *December*

Chidamber, S.R, D.P. Darcy, C.F. Kemerer (1991) "Towards a Metrics suite for Object Oriented Design," *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91),* Sixth Annual Conference, Phoenix, Arizona, USA, October

Chidamber, S.R, D.P. Darcy, C.F. Kemerer (1998) "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Transactions on Software Engineering, 24*(8): 629-639, *August*

Churcher, N., M.J. Shepperd (1995), "Comments on 'A metrics suite for object oriented design,'" *IEEE Transactions on Software Engineering, 21*(3): 263-265, *March*

Cohen, J. (1977) Statistical power analysis for the behavioral sciences, Academic Press, New York

Constantine, L.L, E. Yourdon (1979) Structured design, Prentice-Hall

Cook, T.D., H. Cooper, D.S. Cordray, H. Hartmann, L.V. Hedges, R.J. Light, T.A. Louis, F. Mosteller (1994) Meta-Analysis for explanation - A casebook, Russell Sage Foundation

Daly, J., A. Brooks, J. Miller, M. Roper, M. Wood (1996) "Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software," *Empirical Software Engineering, 1*(2): 109-132

El Emam, K., S. Benlarbi, N. Goel (1999) "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *Technical Report, NRC/ERB-1062, September*

El Emam, K., W. Melo, J.C. Machado (2001) "The prediction of faulty classes using object-oriented metrics," *The Journal of Systems and Software, 56*(1): 63-75

Fagan, M.E. (1976) "Design and code inspections to reduce errors in program development," *IBM Systems Journal, 15*(3): 182-211

Fenton, N.E. (1991) "The Mathematics of Complexity in Computing and Software Engineering," *The Mathematical Revolution Inspired by Computing*, Oxford University Press, pp. 243-256

Fenton, N.E., S.L. Pfleeger (1997) Software Metrics: A Rigorous and Practical Approach, Brooks/Cole Pub Co

Fenton, N.E., M. Neil (1999) "A Critique of Software Defect Prediction Models," *IEEE Transactions on Software Engineering, 25*(5): 675-689, *September/October*

Fenton, N.E., N. Ohlsson (2000) "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Transactions on Software Engineering, 26*(8): 797-814, *August*

Fioravanti, F., P. Nesi (to appear) "Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems," *IEEE Transactions on Software Engineering*

Firesmith, D.G. (1992) Object-oriented requirements analysis and logical design: a software engineering approach, New York: John Wiley & Sons

Gilb, T., D. Graham (1993) Software inspection, Wokingham, England; Reading, Mass.: Addison-Wesley

Gourieroux, C., A. Monfort, A. Trognon (1984) "Pseudo maximum likelihood methods: Applications to Poisson models," *Econometrica, 52*: 701-720

Gray, A.R., S.G. MacDonell (1997) "A Comparison of Model Building Techniques to Develop Predictive Equations for Software Metrics," *Information and Software Technology*

Greene, W.H. (2000) Econometric analysis, New York: Macmillan; London: Collier Macmillan

Halstead, M. (1982) Elements of Software Science, Elsevier, North Holland

Harrison, W. (1994) "Software Metrics and Decision-Making," *The Software QA*, Beaverton, OR, ***1***: 5-12

Harrison, R., S. Counsell, R. Nithi (1998) "Coupling Metrics for Object-Oriented Design," *Proceedings of the 5th International Symposium on Software Metrics*, Bethesda, Maryland

Hayes, W. (2000) "Research Synthesis in Software Engineering: A Case for Meta-Analysis," *Proceedings of the 6th IEEE International Symposium on Software Metrics*, Boca-Raton, Florida, USA

Hedges, L.V., I. Olkin (1985) Statistical methods for meta-analysis, Academic Press, Orlando

Henderson-Sellers, B., L.L. Constantine, I.M. Graham (1996) "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)," *Object Oriented Systems, 3*(3): 143-158

Hitz, M., B. Montazeri (1996) "Chidamber & Kemerer's metrics suite: a measurement theory perspective," *IEEE Transactions on Software Engineering*, ***22***(4): 267-271, *April*

Humphrey, W. (1989) Managing the Software Process, Addison-Wesley, Reading, MA

Hunter, J.E., F.L Schmidt (1995) Methods of Meta-Analysis - Correcting Error and Bias in Research Findings, Newbury Park: Sage Publications

Hu, Q. (1997) "Evaluating Alternative Software Production Functions," *IEEE Transactions on Software Engineering*, ***23***(6): 379-387, *June*

Im, E. (1994) "Unequal Numbers of Observations and Partial Efficiency Gain," *Economics Letters*, ***46***: 291-294

Jackson, M.A. (1983) System development, Prentice-Hall

Jacobson, I. (1992) Object-oriented software engineering: a use case driven approach, Addison-Wesley Publishing Co.

Kamiya, T., S. Kusumoto, K. Inoue, Y. Mohri (1999) "Empirical evaluation of reuse sensitiveness of complexity metrics," *Information and Software Technology*, ***41*** (5): 297-305, *March*

Kearney, J.K., R.L. Sedlmeyer, W.B. Thompson, M.A. Gray, M.A. Adler (1986) "Software Complexity Measurement," *Comm. of the ACM*, ***29***(11): 1044-1050

Khoshgoftaar, T.M., E.B. Allen (1994) "Applications of information theory to software engineering measurement," *Software Quality Journal*, *3*(2): 79-103

Kitchenham, B.A. (2001) "Modeling Software Measurement Data," *IEEE Transactions on Software Engineering*, ***27***(9): 789-804, *September*

Kitchenham, B.A. (1998) "A procedure for analyzing unbalanced data sets," *IEEE Transactions on Software Engineering*, ***24***(4): 278-301, *April*

Kitchenham, B.A., S.L. Pfleeger, N. Fenton (1995) "Towards a Framework for Software Measurement Validation," *IEEE Transactions on Software Engineering*, ***21***(12): 929-944, *December*

Lambert, D. (1992) "Zero-inflated poisson regression with an application to defects in manufacturing," *Technometrics*, **34**: 1-14

Leisch, F., C.J. Lakhmi, K. Hornik (1998) "Cross-Validation with Active Pattern Selection for Neural-Network Classifiers," *IEEE Transactions on Neural Networks*, ***9***(1): 35- 46, *January*

Li, W. (1998) "Another metric suite for object-oriented programming," *Journal of Systems and Software*, ***44***(2): 155-162

Li, W., S. Henry (1993) "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software*, ***23***: 111-122

Liao, Y.K.C. (1998) "Effects on hypermedia versus traditional instruction on students' achievement: A meta-analysis," *Journal of Research on Computing in Education*, ***30***(4): 341

Light, R.J., D.B. Pillemar (1984) "Summing Up: The Science of reviewing research," Harvard University Press, Cambridge, Massachusetts, and London, England Systems," *International Software Engineering Network Technical Report, ISERN-99-16*, pp. 267-271

Lloyd, C.J. (1999) Statistical Analysis of Categorical Data, Wiley-Interscience

Long, J.S. (1997) Regression Models for Categorical and Limited Dependent Variables, Advanced Quantitative Techniques in the Social Sciences, No 7, Sage Publications

Lorenz, M., J. Kidd (1994) Object-Oriented Software Metrics, Object-Oriented Series, Prentice Hall: Englewood Cliffs, N.J.

Marchesi, M. (1998) "OOA metrics for the Unified Modeling Language," *Second Euromicro Conference on Software Maintenance and Reengineering*, Los Alamitos, CA, USA

McCabe, T.J. (1976) "A complexity measure," *IEEE Transactions on Software Engineering*, *2*(4): 308-20

Metamata (2001) JavaCC Documentation, http://download.metamata.com/javaccdocs.zip

Miller, J. (2000) "Can results from Software Engineering experiments be safely combined?" *Proceedings of the 6th IEEE International Symposium on Software Metrics*, Boca-Raton, Florida, USA

Miller, B.K., H. Pei, K. Chenho (1999) "Object-oriented architecture measures," *32nd Annual Hawaii International Conference on Systems Sciences*, Los Alamitos, CA, USA

Miluk, G. (1993) "The Role of Measurement in Software Process Improvement," *Proceedings of the Third International Conference on Software Quality*, Lake Tahoe, Nevada, 4-6 October, pp. 191-197

Mišić, V.B., D.N. Tešić (1997) "Estimation of effort and complexity: An object-oriented case study," *The Journal of Systems and Software, 41*(2): 133-143

Mitchell, T.M. (1997) Machine Learning, New York: McGraw-Hill

Morasca, S., L.C. Briand (1997) "Towards a Theoretical Framework for Measuring Software Attributes," *Proc of the Fourth METRICS'97*, Albuquerque, Nov. 5-7, pp. 119-126

Naur, P., B. Randell, J.N. Buxton (1976) "Software Engineering: Concepts and Techniques," *Proceedings of the NATO Conferences*, London: Mason/Charter Publishers, Inc.

Nesi, P., T. Querci (1998) "Effort estimation and prediction of object-oriented systems," *The Journal of Systems and Software, 42*(1): 89-102

Olkin, I., J. Pratt (1958) "Unbiased estimation of certain correlation coefficients," *Annals of Mathematical Statistics, 29:* 201-211

Papoulis, A. (1991) Probability, Random Variables, and Stochastic Processes, McGraw Hill College Div

Peters, J.F., W. Pedrycz (2000) Software Engineering, An Engineering Approach, John Wiley & Sons

Pfleeger, S.L., L. Hatton, C.C. Howell (2001) Solid Software, Prentice Hall

Pickard, L.M., B.A. Kitchenham, P.W. Jones (1998) "Combining empirical results in software engineering," *Information and Software Technology, 40*(14): 811-821

Pickard, L., B.A. Kitchenham, S. Linkman (1999) "An Investigation of Analysis Techniques for Software Data sets," *In Proceedings of the Sixth International Symposium on Software Metrics (Metrics'99),* Boca Raton FL, USA, IEEE Computer Society Press, pp. 130-142

Porter, A.A., R.W. Selby (1990) "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software*, 7(2): 46-54, *March/April*

Porter, A.A., L.G. Votta (1994) "An Experiment to Assess Different Defect Detection Methods for Software Requirements Inspections," *Sixteenth International Conference on Software Engineering*, Sorrento, Italy

Porter, A.A., L.G. Votta, V.R. Basili (1995) "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," *IEEE Transactions on Software Engineering, 21*(6): 563-575, *June*

Porter, A.A., P.S. Harvey, C.A. Toman, L.G. Votta (1997) "An Experiment to Assess the Cost-Benefits of Code Inspections in Large Scale Software Development," *IEEE Transactions on Software Engineering, 23*(6): 329-346, *June*

Porter, A.A., P.M. Johnson (1997) "Assessing Software Review Meetings: Results of a Comparative Study of Two Experimental Studies," *IEEE Transactions on Software Engineering, 23*(3): 129-145, *March*

Porter, A.A., H. Siy, A. Mockus, L. Votta (1998) "Understanding the Sources of Variation in Software Inspections," *ACM Transactions on Software Engineering and Methodology,* 7(1): 41-79, *January*

Pressman, R.S. (2001) Software Engineering: A Practitioner's Approach, Fifth Edition, McGraw Hill, New York

Quinlan, J.R. (1993) C4.5: programs for machine learning, San Mateo, CA: Morgan Kaufmann Publishers

Reyes, L., D. Carver (1998) "Predicting object reuse using metrics," SEKE '98, *Tenth International Conference on Software Engineering and Knowledge Engineering,* Skokie, IL, USA

Riel, A.J. (1996) Object-Oriented Design Heuristics, Addison-Wesley Pub Co

Ronchetti, M., G. Succi (1999) "Early estimation of software size in object-oriented environments a case study in a CMM level 3 software firm," *Submitted to IEEE Transactions on Software Engineering*

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, S. Blaha (1991) Object-Oriented Modeling and Design: Solutions Manual, Prentice Hall

Russell, M. (1992) "The Mathematics of Measurement in Software Engineering," in Denvir *et. al., Formal Aspects of Measurement. Springer Verlag,* pp. 209-218

Schmidt, P. (1977) "Estimation of Seemingly Unrelated Regressions with Unequal Numbers of Observations," *Journal of Econometrics, 5*: 365-377

Schneider, G.M., J. Martin, W.T. Tsai (1992) "An Experimental Study of Fault Detection In User Requirements Documents," *ACM Transactions on Software Engineering and Methodology, 1*(2): 188-204

Seaman, C.B., V.R. Basili (1997) "An empirical study of communication in code inspections," *In Proc. of the 1997 International Conference on Software Engineering*, Boston, MA, May 17-24

Selby, R.W., A.A. Porter (1988) "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis," *IEEE Transactions on Software Engineering, 14*(12): 1743-1757

Shepperd, M., M. Cartwright (2001) "Predicting with Sparse Data," *IEEE Transactions on Software Engineering, 27*(11): 987-998, *November*

Shepperd, M.J., G. Kadoda (2001) "Using Simulation to Evaluate Prediction Techniques," *In Proceedings of the Seventh International Symposium on Software Metrics* (*Metrics* 2001) London, IEEE Computer Society Press

Shih, T.K., Y.C. Lin, W.C Pai, C-C. Wang (1998) "An Object-Oriented Design Complexity Metric Based on Inheritance Relationships," *International Journal of Software Engineering and Knowledge Engineering, 8*(4): 541-566

Carnegie Mellon University, Software Engineering Institute (1995) The Capability Maturity Model: Guidelines for Improving Software Process, Addison-Wesley, Reading, MA

Sohn, S.Y. (1999) Meta Analysis of Classification Algorithms for Pattern Recognition, *IEEE Transactions on Pattern Analysis and Machine Intelligence, 21*(11): 1137-1144, *November*

Sommerville, I. (2001) Software Engineering, 6th Edition, Addison Wesley

Stone, M. (1974) "Cross validatory choice and assessment of statistical predictions," *Journal of the Royal Statistical Society*, Series B (36): 111-147

Strike, K., K. El Emam, N. Madhavji (2001) "Software Cost Estimation with Incomplete Data," *IEEE Transactions on Software Engineering, 27*(10): 890-908, *October*

Succi, G., L. Benedicenti, C. Bonamico, T. Vernazza (1998) "The Webmetrics Project – Exploiting Software Tools on Demand," *World Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, FL

Succi, G., L. Benedicenti, T. Vernazza (2001) "Analysis of the effects of software reuse on customer satisfaction in an RPG Environment," *IEEE Transactions on Software Engineering, 27*(5): 473-479, *May 2001*

Tang, M.H., M.H. Kao, M.H. Chen (1998) "An Empirical Study on Object-Oriented Metrics," *Proceedings of the 6th IEEE International Symposium on Software Metrics*, Boca-Raton, Florida, USA, November

Travassos, G.H., F. Shull, M. Fredericks, V.R. Basili (1999) "Detecting Defects in Object Oriented Designs: Using Reading Techniques to increase Software Quality," *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '99)

Teologlou, G. (1999) "Measuring object-oriented software with predictive object points," *10th European Software Control & Metrics Conference*, Herstmonceux, England

Wang, Y. (2001) "Formal Description of Object-Oriented Sofware Measurement and Metrics in SEMS," *7th International Conference on Object-Oriented Information Systems, Calgary,* August 27-28

Weyuker, E. (1988) "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering, 14*(9): 1357-1365, *September*

Wood, M., J. Daly, J. Miller, M. Roper (1999) "Multi-method research: An empirical investigation of object-oriented technology," *The Journal of Systems and Software, 48*(1): 13-26

Yand, Y., R. Weber (1990) "An ontological model of an information system," *IEEE Transactions on Software Engineering, 16*(11): 1282 –1292

Yourdon, E. (1989) Structured Walkthroughs, 4th edition, Englewood Cliffs, NJ: Yourdon Press

Zuse, H., P. Bollmann-Sdorra (1992) "Measurement Theory and Software Measures," in: Denvir *et. al., Formal Aspects of Measurement. Springer Verlag*, 1992, pp. 219-259